2005-06

Constant and power-of-2 segmentation algorithms for a high speed numerical function generator

Valenzuela, Zaldy M.

Monterey California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

CONSTANT AND POWER-OF-2-SEGMENTATION ALGORITHMS FOR A HIGH SPEED NUMERICAL FUNCTION GENERATOR

by

Zaldy Valenzuela

June 2005

Thesis Advisor: Jon T. Butler
Second Reader: Phillip E. Pace

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Title (Mix case letters) Constant and Power-of-2 Segmentation Algorithms for a High Speed Numerical Function Generator		**5. FUNDING NUMBERS** N/A	
6. AUTHOR(S) Zaldy M. Valenzuela			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** N/A	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

The realization of high-speed numeric computation is a sought-after commodity for real world applications, including high-speed scientific computation, digital signal processing, and embedded computers. An example of this is the generation of elementary functions, such as $\sin(x)$, e^x and $\log(x)$. Sasao, Butler and Reidel [Ref. 1] developed a high speed numeric function generator using a look-up table (LUT) cascade. Their method used a piecewise linear segmentation algorithm to generate the functions [Ref. 1]. In this thesis, two alternative segmentation algorithms are proposed and compared to the results of Sasao, Butler and Reidel [Ref.1]. The first algorithm is the Constant Approximation. This algorithm uses lines of slope zero to approximate a curve. The second algorithm is the power-of-2-approximation. This method uses $2^i x$ to approximate a curve. The constant approximation eliminates the need for a multiplier and adder, while the power-of-2-approximations eliminates the need for multiplier, thus improving the computation speed. Tradeoffs between the three methods are examined. Specifically, the implementation of the piecewise linear algorithm requires the most amount of hardware and is slower than the other two. The advantage that it has is that it yields the least amount of segments to generate a function. The constant approximation requires the most amount of hardware to realize a function, but is the fastest implementation. The power-of-2 approximation is an intermediate choice that balances speed and hardware requirements.

14. SUBJECT TERMS Numerical Function Generator, Segmentation Algorithms, Piecewise Linear Approximation, Constant Approximation, Power-of-2-approximation			**15. NUMBER OF PAGES** 97
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

CONSTANT AND POWER-OF-2 SEGMENTATION ALGORITHMS FOR A HIGH SPEED NUMERICAL FUNCTION GENERATOR

Zaldy M. Valenzuela
Lieutenant, United States Navy
B.S., University of San Diego, 1999

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2005

Author: Zaldy M. Valenzuela

Approved by: Jon T. Butler
Thesis Advisor

Phillip E. Pace
Second Reader

John P. Powers
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The realization of high-speed numeric computation is a sought-after commodity for real world applications, including high-speed scientific computation, digital signal processing, and embedded computers. An example of this is the generation of elementary functions, such as $\sin(x)$, e^x and $\log(x)$. Sasao, Butler and Reidel [Ref. 1] developed a high speed numeric function generator using a look-up table (LUT) cascade. Their method used a piecewise linear segmentation algorithm to generate the functions [Ref. 1]. In this thesis, two alternative segmentation algorithms are proposed and compared to the results of Sasao, Butler and Reidel [Ref.1]. The first algorithm is the Constant Approximation. This algorithm uses lines of slope zero to approximate a curve. The second algorithm is the power-of-2-approximation. This method uses $2^i x$ to approximate a curve. The constant approximation eliminates the need for a multiplier and adder, while the power-of-2-approximations eliminates the need for multiplier, thus improving the computation speed. Tradeoffs between the three methods are examined. Specifically, the implementation of the piecewise linear algorithm requires the most amount of hardware and is slower than the other two. The advantage that it has is that it yields the least amount of segments to generate a function. The constant approximation requires the most amount of hardware to realize a function, but is the fastest implementation. The power-of-2 approximation is an intermediate choice that balances speed and hardware requirements.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

A special thanks to Jon T. Butler for his support, patience, and insight and for allowing me to work on such a worthwhile study.

To my wife, Joanna, the one that has sacrificed the most, I thank you for your love and support through this process.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The realization of high-speed numeric computation is a sought-after commodity for real world applications, including high-speed scientific computation, digital signal processing, and embedded computers. An example of this is the generation of elementary functions, such as $\sin(x)$, e^x and $\log(x)$. Sasao, Butler and Reidel developed a high speed numeric function generator that was made up of a look-up table (LUT) cascade. Their method used a piecewise linear segmentation algorithm to generate the functions. In this thesis, two alternative segmentation algorithms are proposed and compared to the results of Sasao, Butler and Reidel. The first algorithm is the constant approximation. This algorithm uses lines of slope zero to approximate a curve. The second algorithm is the power-of-2-approximation. This method uses $2^i \times x$ to approximate a curve.

New architectures for a high speed numeric generator stem from these algorithms. The constant approximation architecture utilizes only a segment encoder, and memory needed to store all c_0 coefficients. Using a line with slope zero eliminates two components needed to realize the function, thus making it faster. The power-of-2-approximation architecture utilizes the same components as the piecewise linear approximation, except for the use of a shifter instead of a multiplier. This change allows this architecture to be faster than the original. Experimental results show the need for more memory in the constant approximation, and that the power-of-2-approximation serves as an intermediate solution when looking at the factors of speed, complexity, and number of segments generated. Follow-on work is described that is based on this research.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. SYNOPSIS

The realization of high-speed numeric computation is a sought-after commodity for real world applications, including high-speed scientific computation, digital signal processing, and embedded computers. In the military, high-speed numeric generation is used in sensors, specifically radar, infrared, and visible light applications. An example of this is the generation of elementary functions such as $\sin(x)$, e^x and $\log(x)$. Many functions can be approximated by an iterative approach like CORDIC (Coordinate Rotation Digital Computer) and Newton Raphson methods, but this is slow. A naïve table look-up approach, which uses an enormous amount of memory, is a fast way to generate elementary functions. For this method, the function's values are simply stored in a large memory awaiting an input in order to generate an output. With the advances in memory technology, one might think the use of this method would be practical. Unfortunately, if the desired input and output are on the order of 32 bits, the amount of memory needed to generate the function, 2^{32} bits, becomes unrealizable. The question now becomes, is there a non-iterative, compact, accurate way to generate functions?

An answer to this question was studied by Sasao, Butler and Reidel [Ref. 1]. They showed a way to reduce the amount of memory needed to realize a particular function with a high degree of accuracy with almost the same speed as the naïve method. In their model, they used a look-up table (LUT) cascade to realize a piecewise linear approximation for various functions. The LUT cascade method's accuracy depended on the number of segments and on the approximating function being used and a predefined error ε. In each segment, they used $c_1 x + c_0$ to approximate the function. The coefficients c_1 and c_2 would be stored in memory, ready for use once their particular segment index is generated. A good example of how much memory is saved by using this method can be shown by the following calculation. Assume that it takes 32 segments to generate a function. Since there are 2 coefficients per segment, the total number of values that would have been stored in memory would be 64. This is a much smaller number than 2^{19} bits needed in the naïve method.

With this newly proposed numeric function generator, another question arises? Is there a way to improve on the design? This question was the basis for this thesis, in particular, the improvement of the segmentation algorithm. Chapter II introduces the CORDIC, Newton-Raphson, and Douglas Peuker algorithms. Chapter III introduces and examines the constant and the power-of-2-approximations. In Chapter IV, the new algorithms, the constant approximation and the power-of-2-approximation, are compared to the results derived from piecewise linear method. Chapter V introduces an analytical method for determining the number of segments for each segmentation algorithm is presented as well as a proof is given for determining the optimum segmentation algorithm. Nuances of the power-of-2-approximation are explained. Finally, Chapter VI gives a recommendation stating which segmentation algorithms should be used, and why.

II. NUMERIC FUNCTION GENERATOR ALGORITHMS

In this chapter, the iterative CORDIC, Newton-Raphson, and Douglas-Peucker algorithms will be briefly examined to illustrate some of the past methods used in numeric function generation. This analysis gives insight into why a table look-up method would be advantageous.

A. CORDIC ALGORITHM

Before exploring alternatives for high speed numeric function generators, the accepted implementations should be considered. One of the widely used approximations for generating functions is the CORDIC algorithm. This algorithm uses an iterative process that evaluates functions, such as sine, cosine, and the square root using multiple "pseudo rotational" steps in order to accomplish the goal [Ref. 2]. The generalized CORDIC iteration is described by Parhami in the following equations [Ref. 2]:

$$x^{(i+1)} = x^{(i)} - d_i y^{(i)} 2^{-i} \tag{2.1}$$

$$y^{(i+1)} = y^{(i)} - d_i x^{(i)} 2^{-i} \tag{2.2}$$

$$z^{(i+1)} = z^{(i)} - d_i \tan^{-1} 2^{-i}. \tag{2.3}$$

In the equations, x and y represent endpoints of a vector that is being rotated, and z represents the angle that is being reduced to zero. The variable i represents the number of iterations being used to calculate the approximation for a function. Figure 2.1 depicts the converging nature of the algorithm. An initial guess is implemented, and after several iterations, the error decreases to an acceptable preset value. It is up to the person implementing this algorithm to determine the amount iterations needed to reach an acceptable error ε.

The hardware implementation is fairly simple to realize. Figure 2.2 depicts the essentials needed to realize the machine. Three registers are needed to hold the values of x, y, and z, shifters and ALUs are used to implement the CORDIC equations, and a look-up table is used to store the $\tan^{-1} 2^{-i}$ values generated in memory [Ref. 2]. This method has been proven to be an accepted solution because of the cost required to create

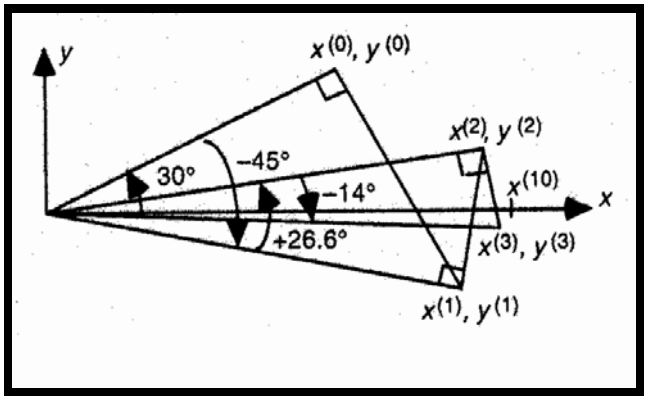


Figure 2.1. CORDIC Algorithm Behavior [From Ref. 2.]

this machine and the speed at which the answer is generated. This method for generating functions is very popular. Lo, Lin and Yang [Ref. 3] used this method to generate sine and cosine functions while Lang and Antelo [Ref. 4] used it to generate the arcsine and arccosine.

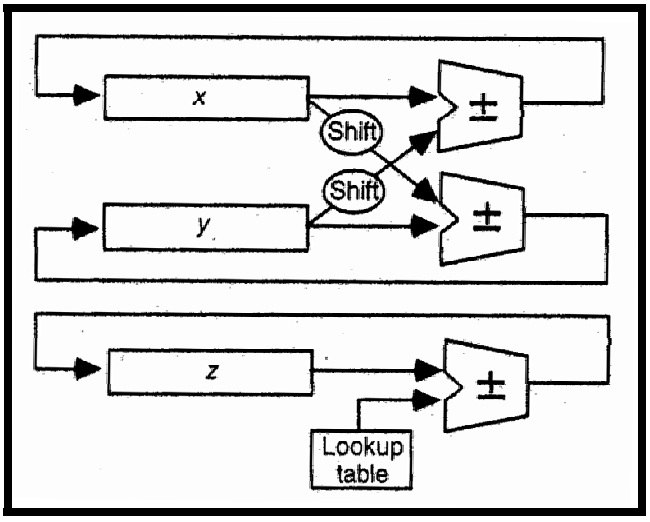


Figure 2.2. CORDIC Hardware Implementation [From Ref. 2.]

B. NEWTON-RAPHSON ALGORITHM

One approach that has been investigated is the use of the Newton-Raphson algorithm for approximating elementary functions such as the sine, cosine, and log functions. The method uses an iterative approach and has been tested specifically for the square root function using a partial product array. Using the Newton-Raphson Method, many equations can be derived as the basis for other elementary functions. The following examples

depict the use of the Newton-Raphson method on some of the commonly studied functions, specifically $\sqrt{x}$ and $1/x$. Agarwal, Gustavson and Schmookler [Ref. 5] used this method to calculate the square root and divide functions as the basis for the Power 3 processor of the Power PC chip.

Example 2.1. Newton Raphson method for $\sqrt{x}$:

$$F\left(x_i\right) = x^2 - A \tag{2.4}$$

$$F'\left(x_i\right) = 2x_i \tag{2.5}$$

$$\tan\theta = F'\left(x_i\right) = \frac{x_i^2 - A}{x_i - x_{i+1}} \tag{2.6}$$

$$2x_i = \frac{x_i^2 - A}{x_i - x_{i+1}} \tag{2.7}$$

$$x_i - x_{i+1} = \frac{x^2 - A}{2x_i} \tag{2.8}$$

$$x_{i+1} = x_i - \frac{x_i^2 - A}{2x_i} \tag{2.9}$$

$$x_{i+1} = x_i - \frac{x_i}{2} + \frac{A}{2x_i} = \frac{x_i}{2} + \frac{A}{2x_i} \tag{2.10}$$

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{A}{x_i}\right) \tag{2.11}$$

Example 2.2. Newton Raphson method for $1/x$:

$$F\left(x_i\right) = \frac{1}{x_i} - A = x_i^{-1} - A \tag{2.12}$$

$$F'\left(x_i\right) = -x_i^{-2} \tag{2.13}$$

$$\tan\theta = F'\left(x_i\right) = \frac{\frac{1}{x_i} - A}{x_i - x_{i+1}} \tag{2.14}$$

$$-x_i^{-2} = \frac{x_i^{-1} - A}{x_i - x_{i+1}} \tag{2.15}$$

$$x_i - x_{i+1} = \frac{x_i^{-1} - A}{-x_i^{-2}} \tag{2.16}$$

$$x_{i+1} = x_i + \frac{x_i^{-1}}{x_i^{-2}} - \frac{A}{x^{-2}} = 2x_i - x^2 A \tag{2.17}$$

Figure 2.3 illustrates the hardware necessary to realize the Newton-Raphson method. There are two inputs into the Newton-Raphson Equation hardware with a feedback loop implemented to realize the iterative nature of the algorithm.

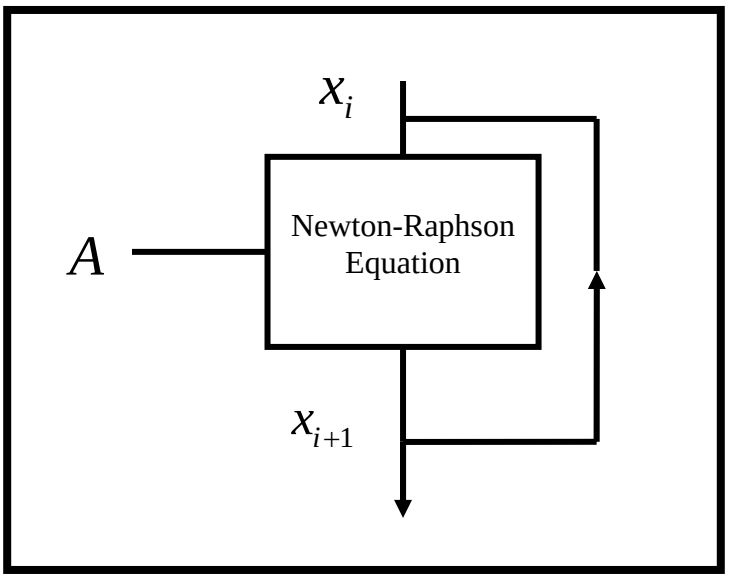


Figure 2.3. Newton Raphson Implementation

C. DOUGLAS-PEUCKER ALGORITHM

The Douglas-Peucker Algorithm was the piecewise linear approximation initially chosen by Sasao, Butler and Reidel [Ref. 1], when developing their idea of a high speed numeric function generator. The algorithm was developed by David Douglas and Thomas Peucker [Ref. 6] and has been effective in such applications as 3-D modeling, cartography, and linear approximation. The algorithm works by calculating the point with the maximum distance from an edge segment [Ref. 7]. Given a line that needs to be sim-

plified, the algorithm starts by joining the starting point and the end point of the line with a

straight line segment [Ref. 7]. It then calculates the perpendicular distance of all vertices from this line [Ref. 7]. If the distance between each vertex and the line segment is within a specified tolerance, the straight line segment represents the whole line in its simplified form [Ref. 7]. If the tolerance condition is not met, the point with the greatest distance from the straight line segment is selected, and the straight line segment is subdivided, joining the two end points to the point of maximum distance [Ref. 7]. This process is repeated until all vertices are within the specified tolerance as this algorithm relies on a recursive decomposition of the line. The following figures illustrate the behavior of the algorithm.

Figure 2.4 depicts the behavior of the algorithm. In stage 1, the algorithm draws a straight line between the endpoints of the curve that is being approximated. In stage 2,

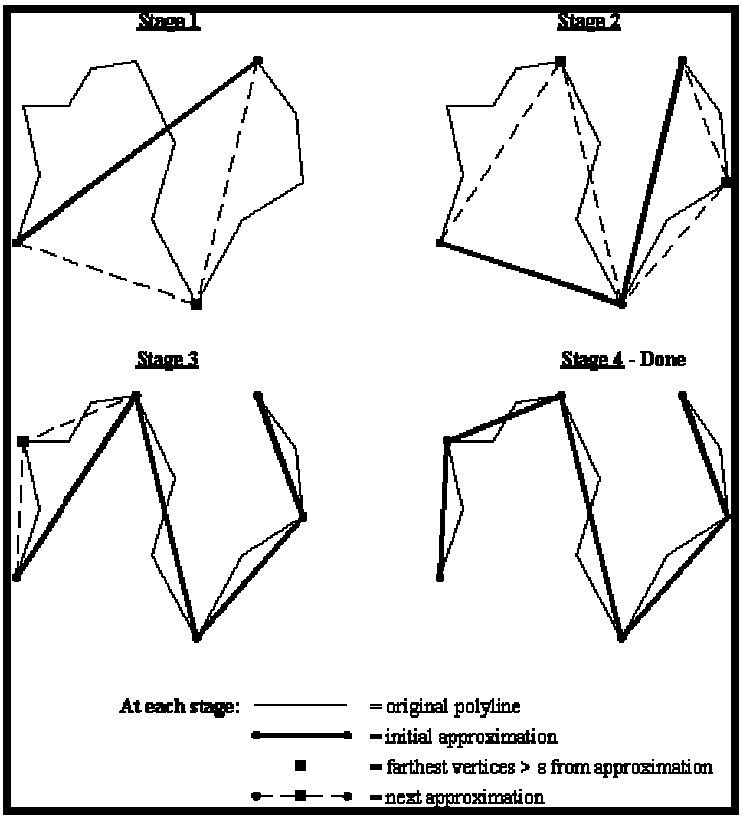


Figure 2.4. Douglas Peucker Algorithm Behavior [From Ref. 7.]

the point farthest from the line is calculated, and then the new approximation is drawn with the first 2 endpoints and the point that is farthest from the line. This process continues until predetermined number of iterations is reached.

D. CHAPTER SUMMARY

In this chapter, three different algorithms used for numeric function generation were examined. The examination of these three algorithms show that the iterative nature of the algorithms and the need for a faster implementation. In the next chapter, three non-iterative algorithms are introduced.

III. THE PROBLEM

In this chapter, three non-iterative algorithms for numeric function generation are presented. The algorithms are the piecewise linear approximation, the constant approximation, and the power-of-2 approximation. Hardware design and basic behavioral examples of each algorithm are presented.

A. PIECEWISE LINEAR APPROXIMATION

The original problem that was investigated by Sasao, Butler and Reidel [Ref. 1] examined the idea of creating both an algorithm and architecture for generating functions in a non-iterative manner. The piecewise linear approximation determined the segmentation of a given function approximating the function by using $c_1 x + c_0$ [Ref. 1]. The process by which the function is generated is dependent on four modules: a segment index encoder, a coefficients table, a multiplier and an adder [Ref. 1]. The purpose of the encoder is to generate a segment index number [Ref. 1]. The segment index number is used to generate the values of c_1 and c_0 that are stored in the coefficients table [Ref. 1]. In order to realize $c_1 x + c_0$, a multiplier is used to generate the term c_1, and an adder is used to combine the first and second terms together. The reason why the architecture for the high speed numeric function generator in Figure 3.1 is reasonable is the assumption that the segment index encoder is reasonably simple and fast [Ref. 1] . With a single segment represented as $c_1 x + c_0$, the total set of lines between a specified interval approximates the desired function.

The key to approximation is the total number of segments needed to realize the function. The total number of segments generated translates to a control word used in a memory. For example, if 32 segments were needed to generate a function, the control word would be 5 bits long ($\log_2 S$, where S is the number of segments) [Ref. 1] . With the ability to encode the segments, the coefficients c_1 and c_0 could be associated to their specific segment.

With its non-iterative nature and it ability to save memory, the piecewise linear approximation seemed to be a reasonable solution. With all these advantages, the machine realization of the algorithm seemed troublesome. The problems with this approach

were the unbounded nature of slopes required to realize a function and the need to utilize a multiplier in the implementation. Any slope, may it be an integer or fraction, could be used in this algorithm, and on the hardware side multipliers are slow. With this in mind, an alternative segmentation algorithm was needed.

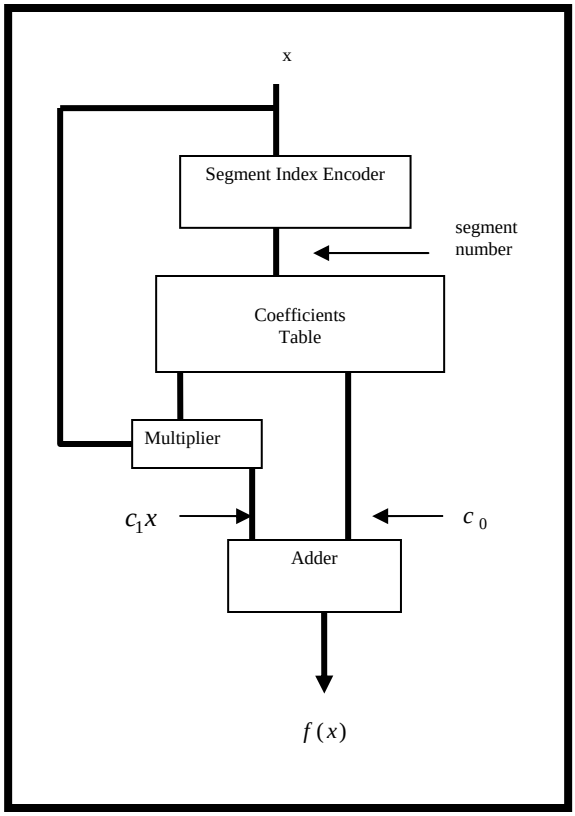
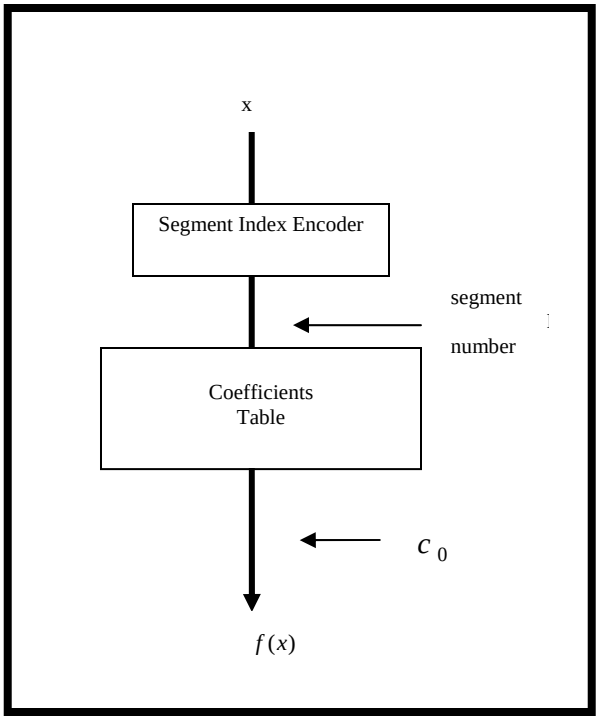


Figure 3.1. Architecture for Numerical Function Generator Using Piecewise Linear Implementation [From Ref. 1.]

B. CONSTANT APPROXIMATION

The first approximation developed to answer the need for an alternative solution was the constant approximation. The constant approximation approximates a given function as a set of step functions. Like the original piecewise linear approximation algorithm, it requires as input a precision. Unlike the piecewise linear approximation, it only requires the output of the numerical function generator to be c_0. The architecture of this numerical function generator is depicted in Figure 3.2. The use of the constant approximation eliminates the multiplier and adder required by the original architecture for the

numerical function generator. The downside to such a change is the increased memory required due to the increased number of segments required to realize the function. The constant segmentation algorithm developed by Jon Butler [Ref. 8] is shown in Figure 3.2. Notice that the algorithm creates a segment that is dependant on the value of the



Constant Segmentation Algorithm
Segmentation of a given function $f(x)$ in the range $x_{low} \le x \le x_{high}$ for Approximation in each segment by the function $c_1 x + c_0$ where c_1=0

Input: Given function $f(x)$, and range $x_{low} \le x \le x_{high}$.
Output: Optimum segmentation specified by low values of segments, x_i. High values are specified by x_{i+1} and, in the case of x_ϕ, the high value is x_{high}, where ϕ is the number of segments

---

1. $i \leftarrow 1$. $x_i \leftarrow x_{low}$.
2. Find the largest Δ_i such that $x_i \le x < x_i + \Delta_i$,
   $c_0 \leftarrow \{\max\{f(x) - c_1 x\} - \min\{f(x) - c_1(x)\}\} = 2\varepsilon$
3. $x_{i+1} \leftarrow x_i + \Delta_i$. $i \leftarrow i+1$.
4. If $x_i < x_{high}$, go to 2.
5. Else if $x_i > x_{high}$, $x_i \leftarrow x_{high}$.
6. Stop

Figure 3.2. Architecture for a Numerical Function Generator Using Constant Approximation and Segmentation Algorithm

precision ε. In Figure 3.3, the function $\cos(\pi x)$ is used as an example to illustrate the Constant Segmentation Algorithm. Figure 3.4 singles out one segment and depicts how ε affects segmentation of the function. In this example, a large $\varepsilon = 2^{-6}$ is used to show the behavior of the program. In a practical application, ε would be much smaller, for example, 2^{-20}. There are two graphs that help describe the algorithm. The vertical lines

represent the end of each segment determined by the algorithm can be seen in the first of the two graphs, while the actual lines used to approximate the $\cos(\pi x)$ function are visible in the second of the two. For the prescribed $\varepsilon = 2^{-6}$, 32 segments are generated.

C. POWER-OF-2-APPROXIMATION

The second segmentation algorithm developed was the power-of-2-approximation. In this algorithm, a piecewise linear approximation of a curve is conducted, with the form of the line being $c_1 x + c_0$, where the coefficient c_1 is a power of

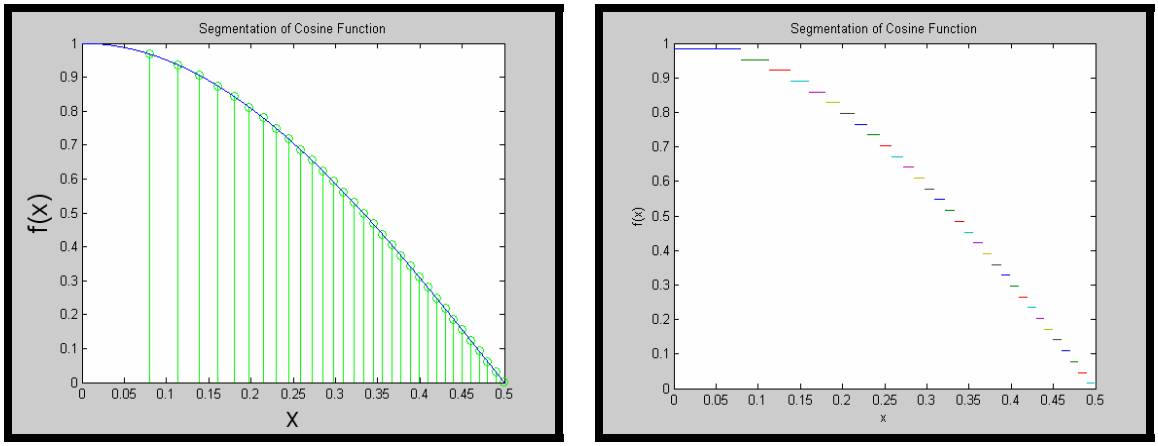


Figure 3.3. Constant Approximation of $\cos(\pi x)$ Between the Interval $0 \le x \le 0.5$

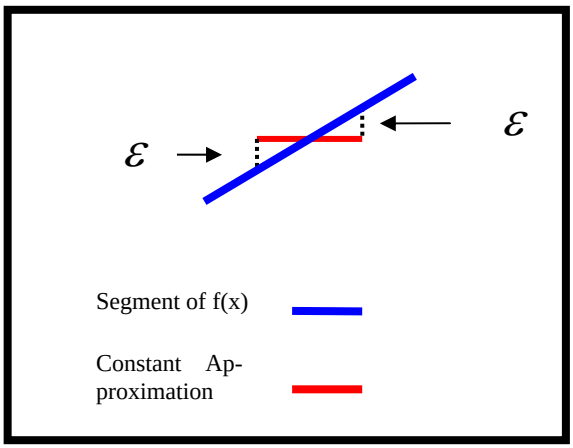
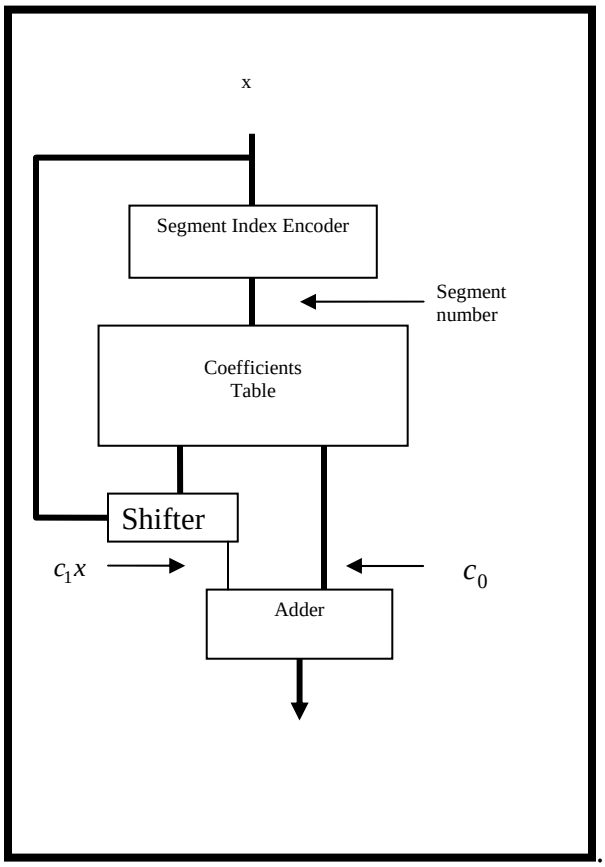


Figure 3.4. The Effects of ε for the Constant Approximation

2. This change is significant because this limits the number of slopes used to realize a function, and it also changes the multiplier in the original architecture of the numerical

function generator into a shifter. This is illustrated in Figure 3.5. Multipliers, in general, are slow when compared to a shifter. The algorithm for this method is shown below. When compared to the constant approximation, the only major difference is the use of $2^i \times x$. Figure 3.6 illustrates the power-of-2 segmentation algorithm. The vertical lines



Power of 2 Segmentation Algorithm

Segmentation of a given function $f(x)$ in the range $x_{low} \le x \le x_{high}$ for Approximation in each segment by the function $c_1x + c_0$ where $c_1 = 2^i$

Input: Given function $f(x)$, finite set S of choices for c_1 in the approximation $c_1x + c_{0,}$ and range $x_{low} \le x \le x_{high}$.
Output: Optimum segmentation specified by low values of segments, x_i. High values are specified by x_{i+1} and, in the case of x_ϕ, the high value is x_{high}, where ϕ is the number of segments

---

1. $i \leftarrow 1$. $x_i \leftarrow x_{low}$.
2. Find the largest Δ_i and a corresponding c_1 such that $x_i \le x < x_i + \Delta_i$,
$\max_{c_1 \in S}\{\max\{f(x) - c_1x\}$
$- \min\{f(x) - c_1(x)\}$
$= 2\varepsilon$
$c_0 \leftarrow \{\max\{f(x) - c_1x\} - \min\{f(x)$
$- c_1(x)\}\}/2$
3. $x_{i+1} \leftarrow x_i + \Delta_i$. $i \leftarrow i + 1$.
4. If $x_i < x_{high}$, go to 2.
5. Else if $x_i > x_{high}$, $x_i \leftarrow x_{high}$.
6. Stop

Figure 3.5. Architecture for Numerical Function Generator Power-of-2-Approximation and Power-of-2 Segmentation Algorithm

represent the end of each segment. The actual lines used to approximate the $\cos(\pi x)$ function are visible in the second of the two graphs. Figure 3.7 singles out one segment and depicts how ε affects the segmentation of the function. For the prescribed $\varepsilon = 2^{-6}$, 7 segments are generated versus 32 when the constant approximation was used. The

number of segments needed to represent the function is decreased. Thus, the number of memory locations needed to store the data also decreases. Although the memory locations needed decreased, the word width increases because of the need to store a constant and a power-of-2 coefficient and constant coefficient are needed. The number of segments needed to represent the function is decreased. Thus, the number of memory locations needed to store the data also decreases. Although the memory locations needed decreased, the word width increased because both a power-of-2 coefficient and constant became necessary.

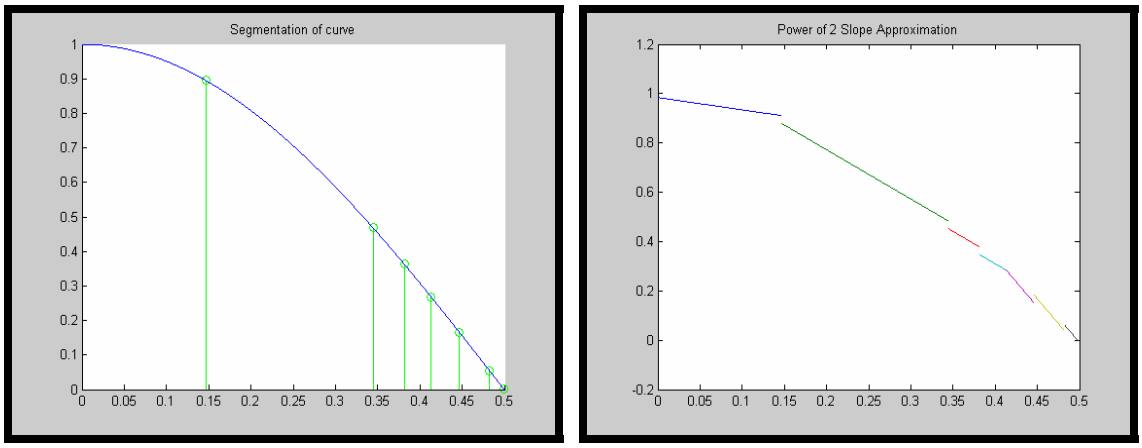


Figure 3.6. Power-of-2-Approximation of $\cos(\pi x)$ Between the Interval $0 \le x \le 0.5$

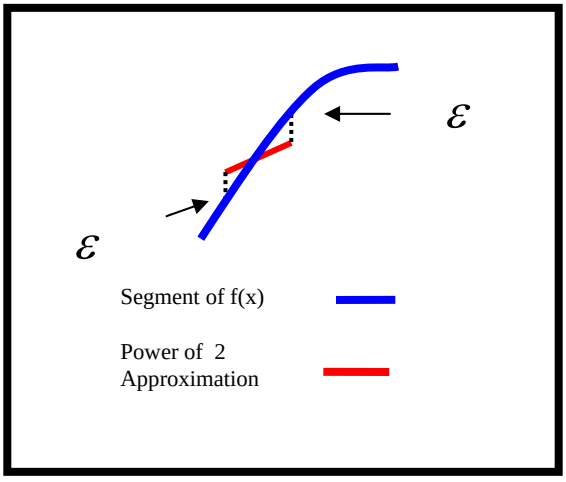


Figure 3.7. The Effects of ε for Power-of-2-Approximation

D. CHAPTER SUMMARY

In this chapter the piecewise linear approximation, constant approximation, and power-of-2 approximation were developed as viable ways of generating numeric functions. In the next chapter, the behavior of the constant and power-of-2 algorithms is examined and comparison data between all three non-iterative algorithms is presented.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. EXPERIMENTAL DATA

In this chapter, the constant approximation and the power-of-2 approximation is examined further, specifically the relationship between the number of segments and the error ε. Comparison results between the two algorithms show the trade-offs between speed versus amount of hardware. Nuances of the power-of-2 approximation are examined. Finally, comparisons are made between the piecewise linear, constant and power-of-2 approximations with respect to numerous functions of interest.

A. PRECISION VERSUS SAMPLES

Before comparative experimental data can be explored, a deeper understanding of the nature of each segmentation algorithm must be examined. For each algorithm, two parameters are vital. The first, is ε, the error associated with the function. The second is the number of samples over which the function is approximated. In the following examples, the ε for $\sin(\pi x)$ is varied in order see how it effects the generation of the functions as well as their behavior.

The functions generated by each algorithm are described by two different figures. The left figure in the pair depicts the edges of the segments, while the right shows the actual curve generated by the approximation. Note that these results are based on 200,000 samples within the period from 0 to 0.5 and the ε ranging from 2^{-7} to 2^{-9}. Figure 4.1

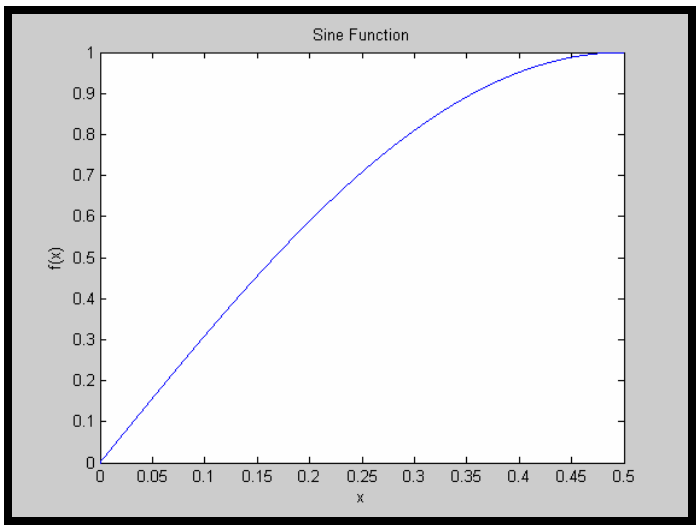


Figure 4.1. MATLAB-Generated $\sin(\pi x)$ Between the Interval $0 \le x \le 0.5$

is a MATLAB-generated $\sin(\pi x)$ function between the interval $0 \le x \le 0.5$. The first comparisons that will be made will be between the MATLAB $\sin(\pi x)$, and the functions generated by the constant approximation. From the outset, the constant approximation with $\varepsilon = 2^{-7}$ (Figure 4.2), is not as accurate when compared the MATLAB-generated

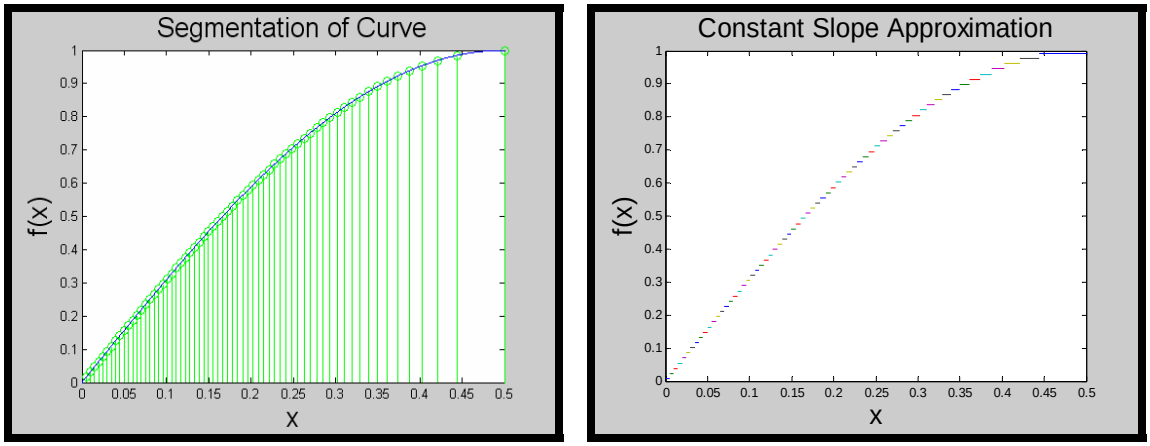


Figure 4.2. Constant Approximation of $\sin(\pi x)$ Between the Interval $0 \le x \le 0.5$ with $\varepsilon = 2^{-7}$

function. Although the generated curve is not accurate, the behavior of the approximation can be readily seen. In Figure 4.3, ε is decreased to 2^{-8}. When comparing Figures

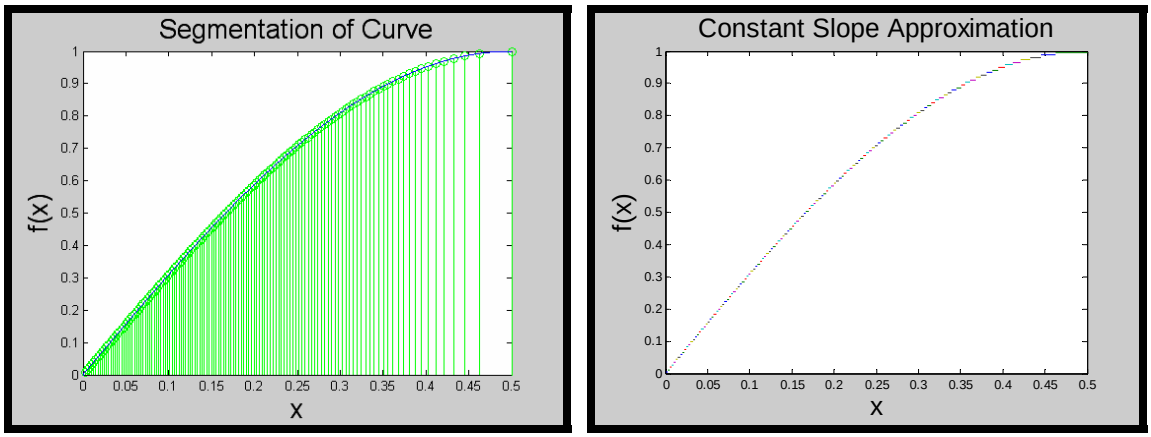


Figure 4.3: Constant Approximation of $\sin(\pi x)$ Between the Interval $0 \le x \le 0.5$ with $\varepsilon = 2^{-8}$

4.2 and 4.3, it become evident that as ε decreases, the number of segments increases and the approximation become more accurate. Figure 4.4 continues to show the trend of increased accuracy with decreased ε.

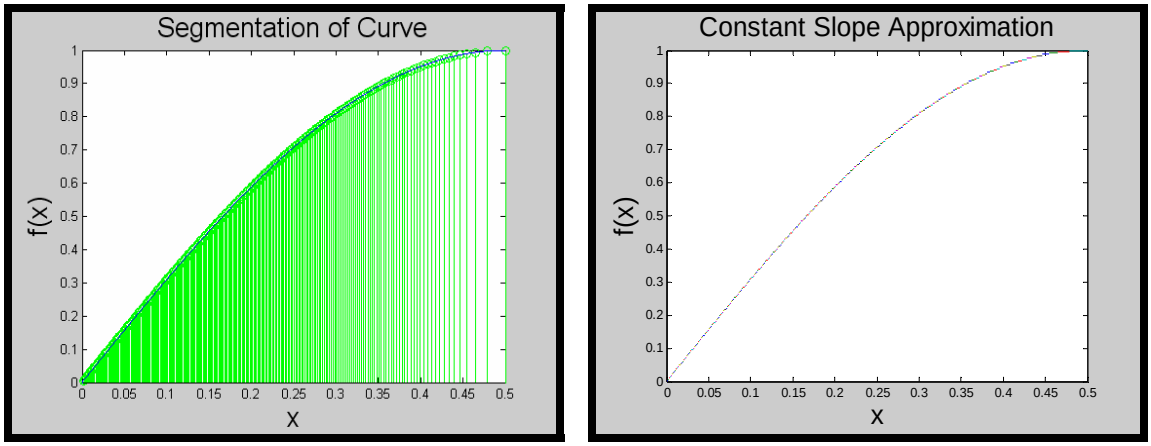


Figure 4.4. Constant Approximation of $\sin(\pi x)$ Between the Interval $0 \le x \le 0.5$ with $\varepsilon = 2^{-9}$

When comparing the power-of-2-approximation to the MATLAB-generated function, the lack of precision of the power-of-2-approximation is evident. When examining Figure 4.5 the behavior of the curve becomes evident. Various powers-of-2 slopes are

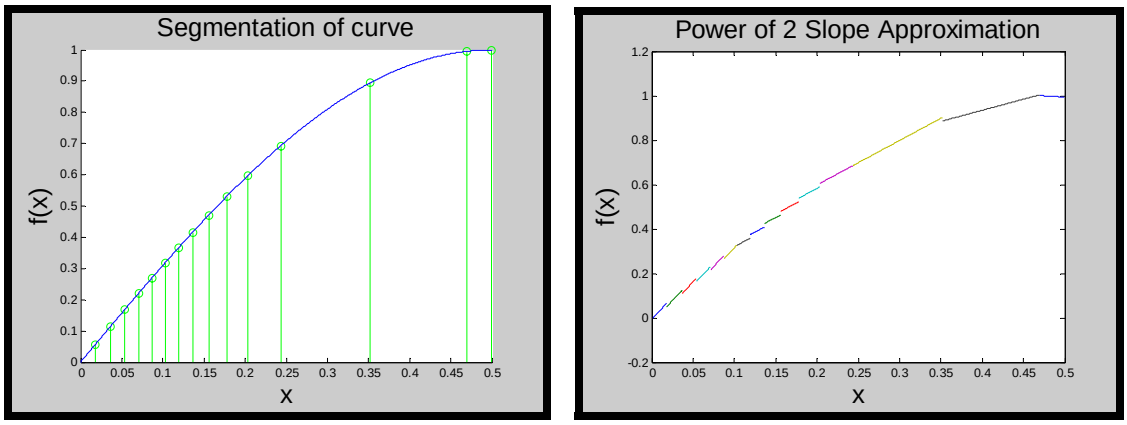


Figure 4.5. Power-of-2-approximation of $\sin(\pi x)$ Between the Interval $0 \le x \le 0.5$ with $\varepsilon = 2^{-7}$

used to approximate the segmentation. Figures 4.6 and 4.7 show that as ε is decreased, the number of segments increases and the approximation of the curve approaches that of

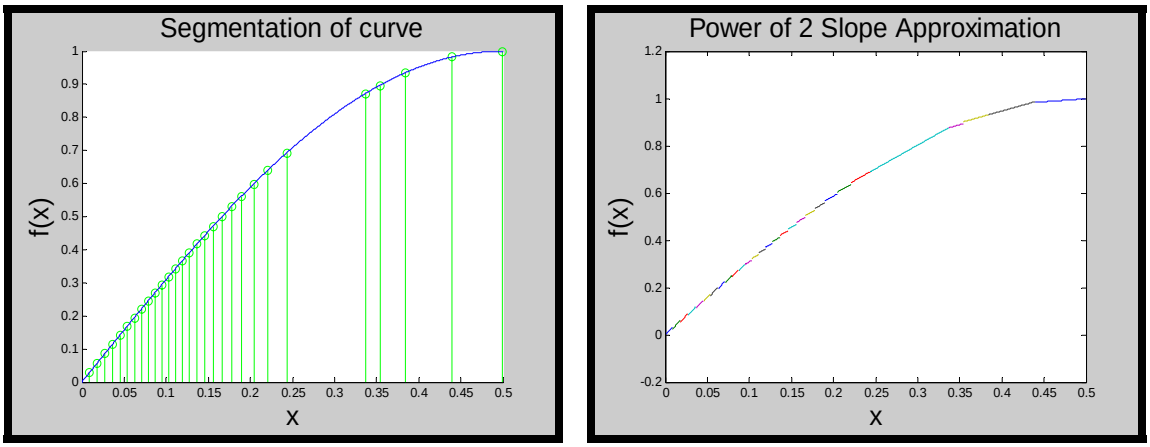


Figure 4.6. Power-of-2-Approximation of $\sin(\pi x)$ Between the Interval $0 \le x \le 0.5$ with $\varepsilon = 2^{-8}$

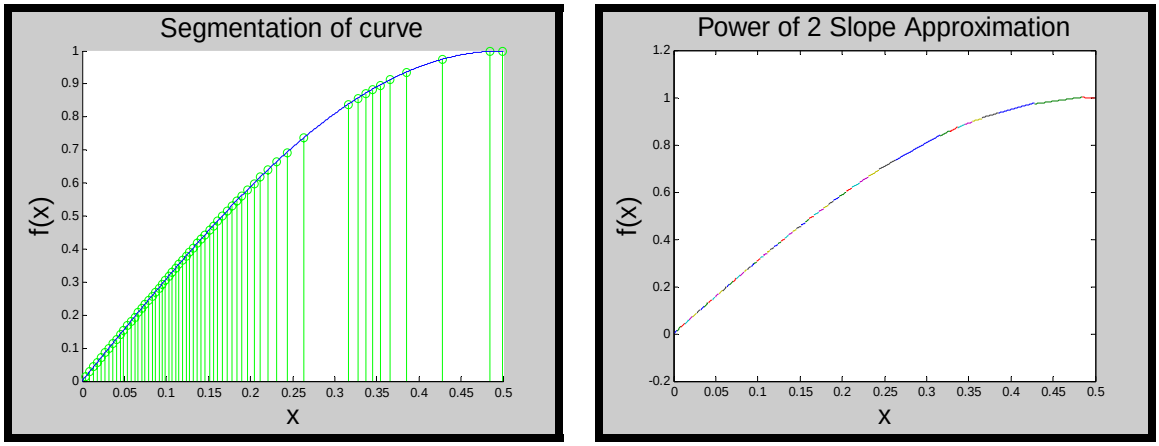


Figure 4.7. Power-of-2-Approximation of $\sin(\pi x)$ Between the Interval $0 \le x \le 0.5$ with $\varepsilon = 2^{-9}$

$\sin(\pi x)$. Table 4.1 compiles the results from the experiments and show that the power-of-2-approximation improves on the constant approximation by about 4.

PRECISION	POWER-OF-2 SEGMENTS	CONSTANT SEGMENTS
2^{-7}	15	64
2^{-8}	29	128
2^{-9}	59	256

Table 4.1. Precision and Segment Count

B. POWER-OF-2-APPROXIMATIONS NUANCES

From the initial numbers taken when comparing the power-of-2-approximation to the constant approximation, the number of segments required to realize a particular function stand out. Decreasing the amount of memory would seem to yield success. For the most part, this is true. Unfortunately, there are some drawbacks associated with the Approximation. One example is shown in Figure 4.5, specifically the tail end of the generated function. Notice, that a segment with a negative slope is generated for the last segment. This would seem to be impossible because when looking at the sine curve between 0 and 0.5, all the slopes are all positive or zero toward the end. The reason this occurs is because multiple power-of-2 slopes yield the same size for the segmentation. When a segment is very small, there appears to be a propensity for that segment to have multiple slopes with that same width.

The error where multiple power-of-2 slopes yield the same width also occur at points where an approximation is changing from one power-of-2 slope to another. This area, which is described as a transition area, can be calculated. Note that this phenomena occurs more frequently when ε is small and the number of samples used to realize a function is approximately 10^6. The MATLAB algorithm was developed to detect segments that have multiple slopes. An example of this is when ε is 2^{-13}, the number of samples is 15 million, and the function is $\sin(x)$. For this case, there are three segments with multiple slopes that yield the same accuracy. Two segments were found to have 2 slopes and the third had six slopes. The first two were located at the transition point from the slope of 2 to 4, and the third was the last segment of the segmentation.

C. COMPARSION DATA

The sine function has been examined to show the differences between the two segmentation algorithms. Other interesting functions, such as $\sqrt{x}$, $1/x$, and the entropy function have been explored. Sasao, Butler, and Riedel [Ref. 1] tested their algorithm on numerous functions and documented their results. In Tables 4.2 and 4.3, their results are compared

FUNCTION	INTERVAL	PIECEWISE LINEAR	POWER-OF-2	CONST	ε
2^x	$[0,1]$	7	42	256	2^{-9}
$\frac{1}{x}$	$[1,2)$	8	44	256	2^{-10}
$\sqrt{x}$	$\left[\frac{1}{32},2\right]$	18	26	159	2^{-8}
$\frac{1}{\sqrt{x}}$	$[1,2)$	4	25	150	2^{-10}
$\log_2(x)$	$[1,2)$	5	44	256	2^{-9}
$\ln(x)$	$[1,2)$	7	31	178	2^{-9}
$\sin(\pi x)$	$\left[0,\frac{1}{2}\right]$	9	59	256	2^{-9}
$\cos(\pi x)$	$\left[0,\frac{1}{2}\right]$	9	57	256	2^{-9}

Table 4.2. Comparison Table of the Three Algorithms

against the results from the approximations presented in this thesis. Figures 4.8 - 4.10 show the segmentation of the aforementioned functions with ε equal to the values de-

scribed in the tables below. Note that a complete compilation of all functions listed in Tables 4.2 and 4.3 is located in Appendices A and B.

FUNCTION	INTERVAL	PIECEWISE LINEAR	POWER-OF-2	CONST	ε
$\tan(\pi x)$	$\left[0,\frac{1}{4}\right]$	8	46	256	2^{-9}
$\sqrt{-\ln(x)}$	$\left[\frac{1}{32},1\right)$	26	40	238	2^{-8}
$\tan^2(\pi x)+1$	$\left[0,\frac{1}{4}\right]$	16	45	256	2^{-9}
$-x\log_2(x)-(1-x)\log_2(1-x)$	$\left[\frac{1}{256},\frac{255}{256}\right]$	28	79	492	2^{-9}
$\frac{1}{1+e^{-4x}}$	$[0,1]$	8	33	247	2^{-10}
$\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$	$\left[0,\frac{1}{2}\right]$	2	4	4	2^{-10}

Table 4.3. Comparison Table of the Three Algorithms

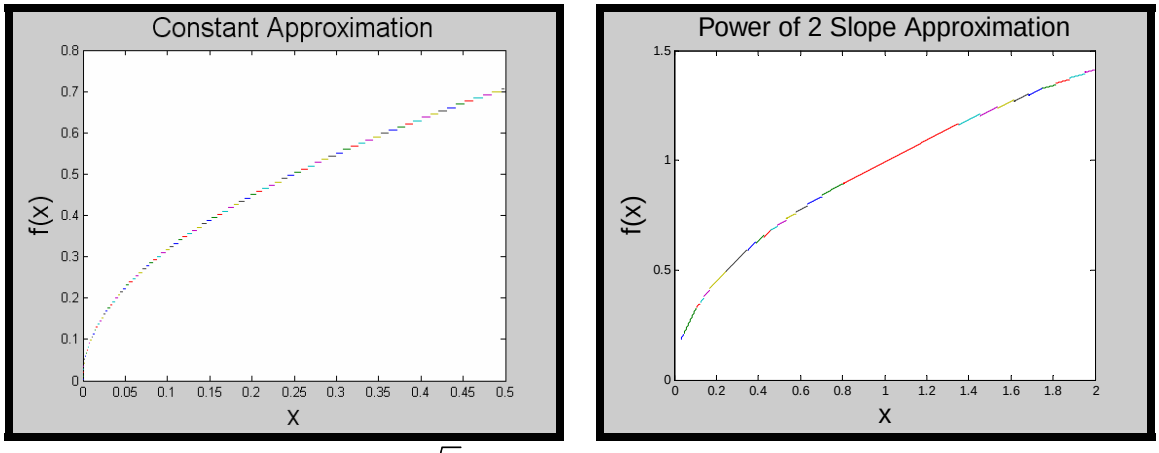


Figure 4.8. $\sqrt{x}$ Approximation by Both Algorithms

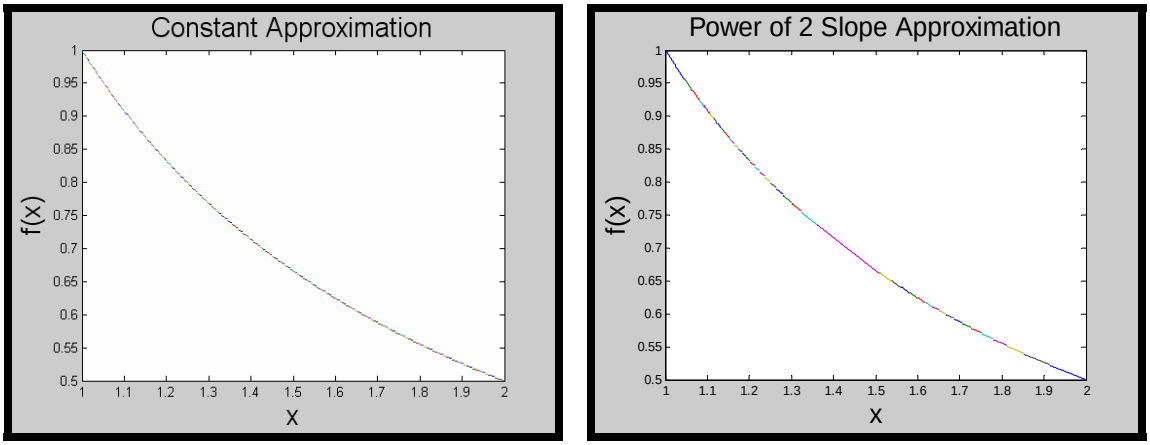


Figure 4.9. $1/x$ Approximation by Both Algorithms

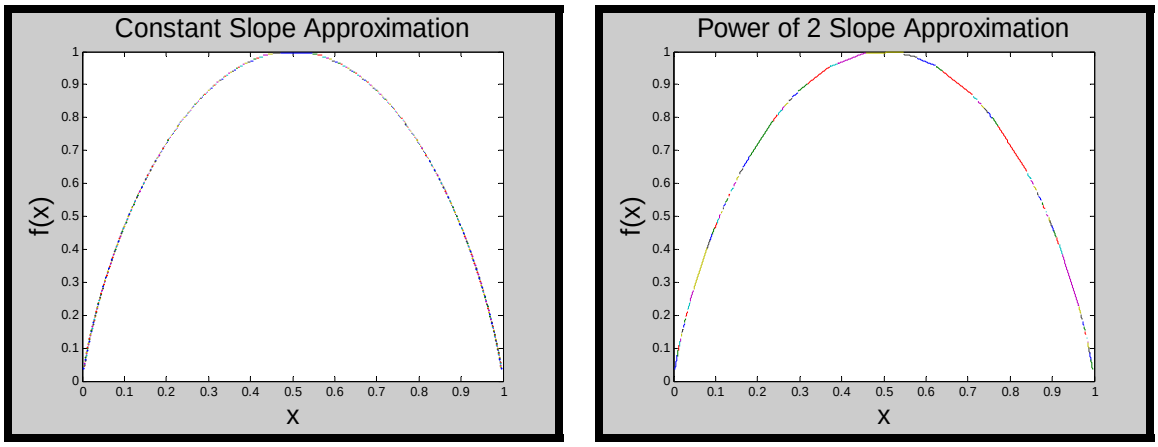


Figure 4.10. Entropy Approximation by Both Algorithms

D. CHAPTER SUMMARY

In this chapter, in depth data was taken from experiments that showed the effects of ε on the segmentation and accuracy of both the constant and power-of-2 approximations. Nuances of the power-of-2 approximation were investigated to show possible problems with the algorithm. The overall comparison between the three non-iterative algorithms in Tables 4.2 and 4.3 lead to results and findings that are expanded on in the following chapter.

V. RESULTS

In this chapter, results are derived from the experimental results in Chapter 4. An analytical method for calculating the segments for the constant approximation is introduced. A connection between the constant and power-of-2 approximations is examined. A table calculating the number of power-of-2 slopes needed to realize a curve shows that not all of them are needed to realize a particular function. This implies a reduction in hardware for the algorithm.

A. CONSTANT APPROXIMATION RESULTS

From the tabulated results, several theorems and lemmas were created by Jon Butler to determine closed form equations for the results determined in Tables 4.2 and 4.3. Butler created theorems and lemmas for both the constant and power-of-2-pproximations.

One of the surprising results from Tables 4.2 and 4.3 was that many of the functions had the same number of segments. Tables 4.2 and 4.3 showed that many functions required 256 segments in order to be realized. At first, it was hypothesized that the intervals of the functions had a direct correlation to this. This was partly true, but, the determining factor-was the behavior of the curve being examined. Butler describes the functions within the prescribed intervals as monotone [Ref. 8]. His definition of a monotone function is based on whether a functions slope is either positive or negative within an entire interval. If a function is monotone, the number of segments can be calculated by the following Lemma.

Lemma 5.1. Let $f(x)$ *be monotone over some interval* $x_{low} \le x \le x_{high}$, *and let* $\Delta = \max\{f(x)\} - \min\{f(x)\}$. *An approximation of* $f(x)$ *using constant functions in each segment accurate to within* ε *requires no more than* $\Delta / 2\varepsilon$ *segments.* [Ref. 8]

Example 5.1.
Take $\sin(\pi x)$, $\varepsilon = 2^{-9}$, and the experimental value of 256 segments from Figure 4.3 and apply Lemma 1.

$$\max\{f(x)\} = \sin(\pi/2) = 1$$
$$\min\{f(x)\} = \sin(0)\} = 0 \qquad (5.1)$$
$$\Delta = 1$$
$$\text{number of segments} = \frac{1}{(2)(2^{-9})} = 256.$$

If a function is not monotonic, for example, $\sin(\pi x)$ in the interval [0,2], that function can be separated into monotonic functions. Lemma 1 can be applied to each monotone region. Figure 5.1 depicts the separation of the $\sin(\pi x)$ function. From here Butler's Theorem 1 can be applied.

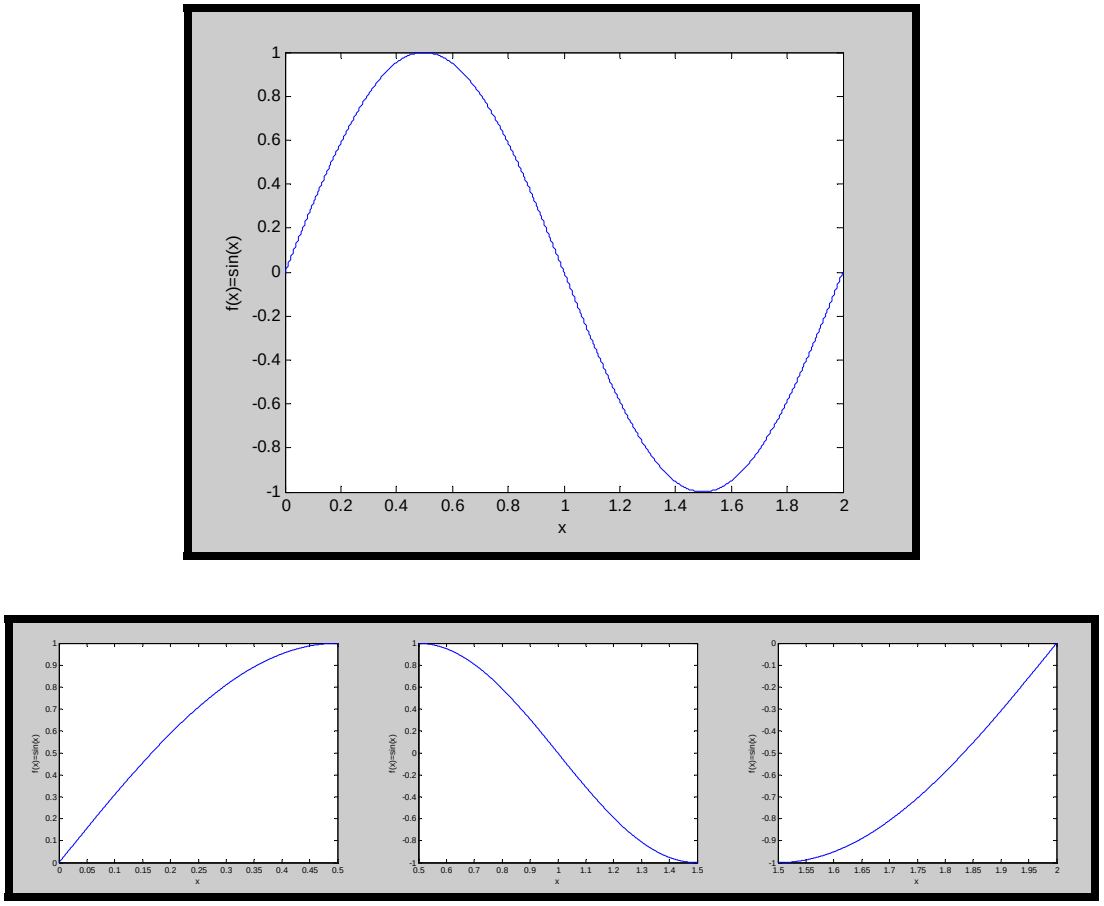


Figure 5.1. Sine Function Separation

Theorem 5.1. Let $f(x)$ be differentiable over some interval $x_{low} \le x \le x_{high}$, and let there be S sections, indexed by i for $1 \le i \le S$, over which $f(x)$ alternates between positive monotone and negative monotone. For all x in Section i, let $\Delta_i = \max\{f(x)\} - \min\{f(x)\}$. An optimum approximation of $f(x)$ using constant functions in each segment accurate to within ε has [Ref. 2]

$$\sigma\left(f\left(x\right)\right)=\sum_{i=1}^{S}\left\lceil\frac{\Delta_i}{2\varepsilon}\right\rceil-S+1 \tag{5.2}$$

segments.

Also, if a function is separated the way it is in Figure 5.1, the last segment and the first segment of adjacent sections can be added together as one segment when calculating the overall segmentation. Through experiments, the total number of segments needed to realize the $\sin(\pi x)$ is 1022. When Theorem 1 is applied, the following answer is determined:

Example 5.2.

All results from Lemma 1.
Number of segments of first Section : 256
Number of segments of second Section: 512
Number of segments of third Section : 256

$$256+512+256=1024 \tag{5.3}$$

since adjacent sections have segments that can be combined, the total number of segments is $1024-2=1022$

B. POWER-OF-2-APPROXIMATION RESULTS

One of the results that came from the Power-of-2-approximation was that the algorithm was also based on the Constant Approximation. Butler developed a Lemma that connected the two approximations together.

Lemma5. 2. Let $f(x)$ *be a differentiable function over some interval* $x_{low}\le x\le x_{high}$. *An optimal segmentation of* $f(x)$ *using* $2^i x+c_0$, *for fixed* i, *is an optimal segmentation of* $F(x)=f(x)-2^i$, *using constant approximation.* [Ref. 2]

Lemma 2 states that the optimal segmentation of any $f(x)$ is based on two things. The first is a subtraction operation between the original function from the desired power-of-2 slope. The second is using c_0, now known as $F(x)$, and applying the constant approximation on it in order to find the segmentation.

C. PARTICULAR SLOPE FOR PARTICULAR FUNCTIONS

While conducting the experiments on the various functions, it became clear that there were certain slopes that were being used more than others while approximating a

curve. The algorithm that was developed looked at slopes that ranged from 2^{-7} to 2^{7}, and also looked at slopes that were small in magnitude, -2^{-7} to 2^{-7}. The algorithm spanned all of the slopes to determine the best one within the given precision. An example of this phenomena is the data generated from the sine curve. With an $\varepsilon = 2^{-9}$, none of the negative slopes and powers of 2 from 2^{3} to 2^{7} were used. Since the range of viable slopes is within a certain range, the algorithm for this particular function could be modified to include only those select slopes. This would reduce the amount of time to generate the function and the amount of memory needed for this kind of machine. If the goal is to make high speed numeric function generators that generate only one specified-function, the values of the slopes used by each function, found in Table 5.1 and 5.2, could help in the design of that machine. Another driving factor in this is the importance of ε.

FUNCTION	**INTERVAL**	**POWER-OF-2 SEGMENTS**	ε	**POWER-OF-2 SLOPES USED**
2^{x}	$[0,1]$	42	2^{-9}	2^{-1}, 2^{0}
$\frac{1}{x}$	$[1,2)$	44	2^{-10}	-2^{0}, -2^{-1}, -2^{-2}
$\sqrt{x}$	$\left[\frac{1}{32},2\right]$	26	2^{-8}	2^{1}, 2^{0}, 2^{-1}, 2^{-2}
$\frac{1}{\sqrt{x}}$	$[1,2)$	25	2^{-10}	-2^{-7}, -2^{-6}, -2^{-5}, -2^{-4}, -2^{-3}, -2^{-2}, -2^{-1}, -2^{0}
$\log_2(x)$	$[1,2)$	44	2^{-9}	2^{0}, 2^{-1}, 2^{-2}
$\ln(x)$	$[1,2)$	31	2^{-9}	2^{0}, 2^{-1}
$\sin(\pi x)$	$\left[0,\frac{1}{2}\right]$	59	2^{-9}	2^{2}, 2^{1}, 2^{0}, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-6}, 2^{-7}

Table 5.1. Specific Slopes for Specific Functions #1

Greater precision with respect to ε yields more slopes within the smaller range (-2^{-7} to 2^{-7}). The converse is also true, that is, if the ε is high, then smaller slopes will not be used to approximate the function.

FUNCTION	INTERVAL	POWER-OF-2	ε	POWER-OF-2 SLOPES USED
$\cos(\pi x)$	$\left[0,\frac{1}{2}\right]$	57	2^{-9}	$-1(2^{2},2^{1},2^{0},2^{-1},2^{-2},2^{-3},2^{-4},2^{-6},2^{-7})$
$\tan(\pi x)$	$\left[0,\frac{1}{4}\right]$	46	2^{-9}	$2^{2},\ 2^{3}$
$\sqrt{-\ln(x)}$	$\left[\frac{1}{32},1\right)$	40	2^{-8}	$-2^{0},-2^{1},-2^{2},-2^{3},-2^{5}$
$\tan^{2}(\pi x)+1$	$\left[0,\frac{1}{4}\right]$	45	2^{-9}	$2^{-2},2^{0},2^{1},2^{2},2^{3},2^{4}$
$-x\log_{2}x(1-x)\log_{2}(1-x)$	$\left[\frac{1}{256},\frac{255}{256}\right]$	79	2^{-9}	$2^{3},2^{2},2^{1},2^{0},2^{-1},2^{-5},-2^{-2},-2^{-1},-2^{0},-2^{1},-2^{2},-2^{3}$
$\frac{1}{1+e^{-4x}}$	$[0,1]$	33	2^{-10}	$2^{0},2^{-1},2^{-2},2^{-3}$
$\frac{1}{\sqrt{2\pi}}e^{\frac{-x^{2}}{2}}$	$\left[0,\frac{1}{2}\right]$	4	2^{-10}	$2^{-5},2^{-4},2^{-3}$

Table 5.2. Specific Slopes For Specific Functions #2

D. POSSIBLE PROBLEMS WITH ALGORITHMS

One of the possible problems with both of these algorithms is the fact that the approximations are made up of lines that are discontinuous. Compared to the Douglas-

Peucker algorithm, where the approximation is continuous, these discontinuities could be a problem, especially if an application were needed to differentiate the approximation.

E. CHAPTER SUMMARY

In this chapter, an analytical expression for the calculation of monotonic curves using the constant approximation was formed. A connection between the constant and power-of-2 approximations was made. Finally, a hardware implementation was proposed for the power-of-2 approximation where only a limited number of slopes could be used to approximate a given curve. In the following chapter, follow-on work is discussed.

VI. FOLLOW-ON WORK AND CONCLUSION

In this chapter, follow-on work is discussed. The hardware implementation of the proposed high speed numeric function generators is one follow-on project. A hardware comparison between different numeric function generators is another. The creation of another segmentation algorithm is also a possibility.

A. SEGMENT INDEX ENCODER IMPLEMENTATION

One of the challenges in the development of the architectures represented in this thesis is the complexity of the segment index encoder. Sasao, Butler, and Reidel [Ref. 1] develop the idea of using a LUT cascade to create the encoder. They state the circuit is a simple circuit as shown in the following theorem:

> *Theorem 6.1: If the segment index function g(x) maps to at most p segments, then there exists a LUT cascade realizing g(x), where the number of interconnecting lines between LUTs is at most* $(\log_2 p)$. [Ref. 1]

Keeping all this in mind, the development of a working segmentation encoder would be the next step in the process of realizing the architectures discussed in the thesis.

B. HARDWARE COMPARISON

The next step in this process would be to test all three architectures against each other, and determine which one would be the optimum machine. Speed, complexity, and segmentation could be examined and be compared to the experimental results documented in this thesis. Another test would be test all three against the iterative CORDIC, Newton-Raphson, and Douglas-Peucker implementations in order truly determine which one is faster.

C. SEGMENTATION ALGORITHM

One of the goals by Sasao, Butler and Reidel [Ref. 1], for their numeric function generator was the ability to realize non-continuous functions such as the step, saw tooth, etc. The data collected from their research, and data from this thesis only looks at function that are continuous. If a function approaches $\pm\infty$, for example, $1/x$ when $x = 0$, the algorithms developed for the constant approximation and power-of-2-approximation are unable to calculate the segmentation. A software or hardware adjustment for identifying discontinuities should be implemented to deal with these cases. For some of the func-

tions, the periodic but discontinuous $\tan(\pi x)$, it would be possible to approximate the curve in one interval, and copy that approximation onto a larger interval to generate the function. The functions that are not periodic have to be dealt with.

There could be an alternative segmentation algorithm that could yield better results than the ones discussed in this thesis. One of the approaches that could be studied is the use of a higher order approximation. This would entail implementing an approximation of a function using x^2 [Ref. 9]. It is hypothesized that the number of segments needed to approximate a curve would be decreased, but the implementation of this algorithm would increase the complexity of the hardware as shown in Figure 6.1. The x^2 portion of the polynomial would require the use of one more multiplier to approximate a function. There seems to be a point of diminishing returns with this particular implementation, but further study could yield better results than expected.

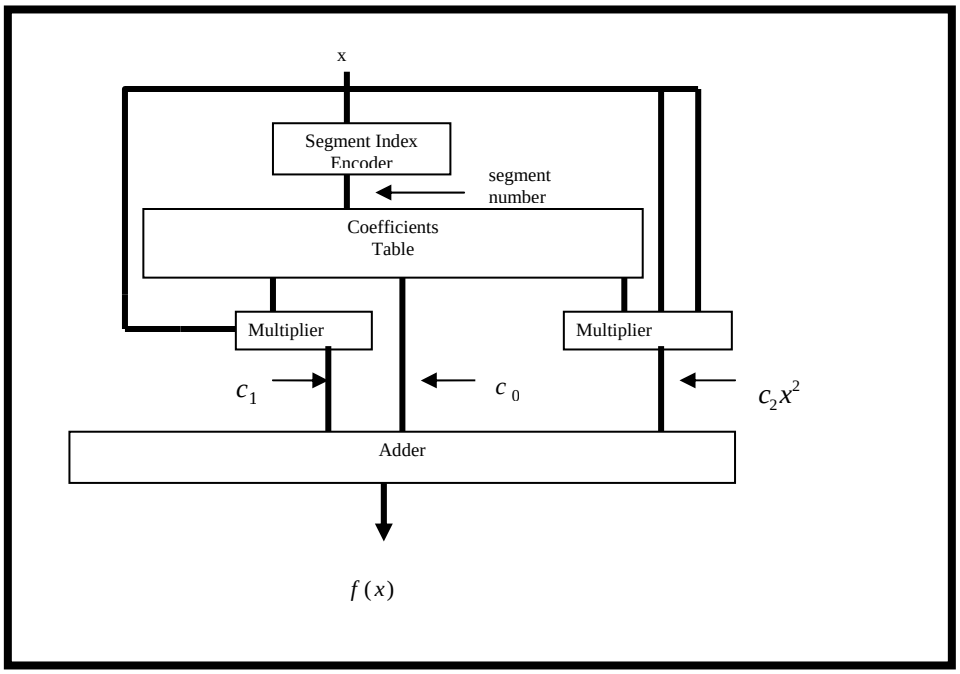


Figure 6.1. Quadratic Approximation Architecture

Another algorithm that could be developed is a hybrid of the power-of-2-approximation where the slopes are not limited to just powers of 2. Notice that any integer value for a slope could be created by adding different power-of-2 slopes together.

For example, if the slope of 3 were needed to be realized, take $2^0 + 2^1$. This hybrid approach would require additional hardware to deal with the slopes desired.

D. CONCLUSION

Comparing the piecewise linear, constant, and power-of-2-approximations, and attempting to determine which one is the "best", is a study in trade offs. Assuming that the LUT Cascade segment encoder is relatively simple, all three show definite promise. With the piecewise linear approximation, the amount of segments needed to approximate is the smallest. Hardware wise, it may is the slowest because of the multiplier in its architecture. The constant approximation is the fastest implementation, but the memory needed in the architecture is the most. The middle of the road solution is the power-of-2-approximation. The number of segments needed to realize a function, and the hardware implementation needed is a compromise between the two extremes. One could argue that the constant approximation may actually be more viable because of the inexpensive nature of memory and the amount available for implementation. One of the drawbacks to this idea is that the number of segments required to approximate a function is also equal to the number of discontinuities in the approximation. If a choice were made between the three approximations, the power-of-2-approximation has the potential of being the best choice. When considering complexity, segment size, speed, and optimum nature, it becomes clear that the power-of-2-approximation would make the best choice.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. CONSTANT APPROXIMATION DATA

The following graphs show the constant approximation of all of the functions listed in Tables 4.2 and 4.3. Note that the number of samples within the interval are 200,000.

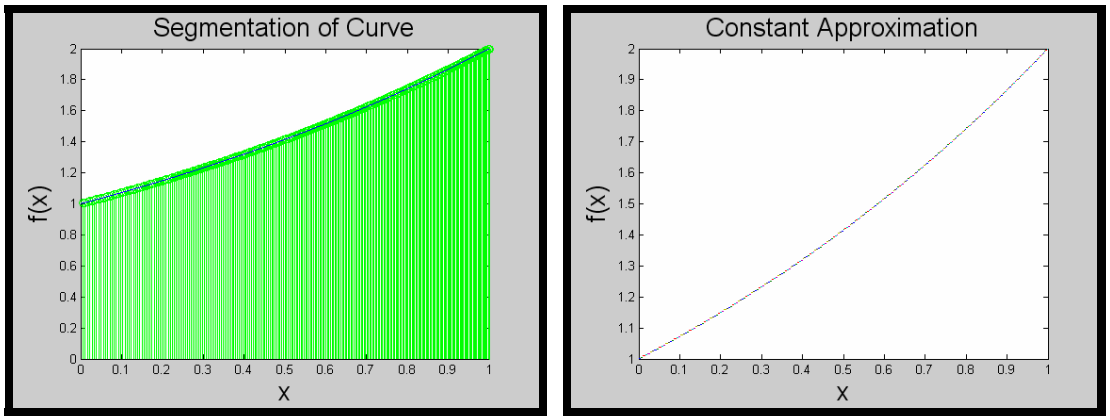


Figure A.1. Constant Approximation of 2^x, $\varepsilon = 2^{-9}$

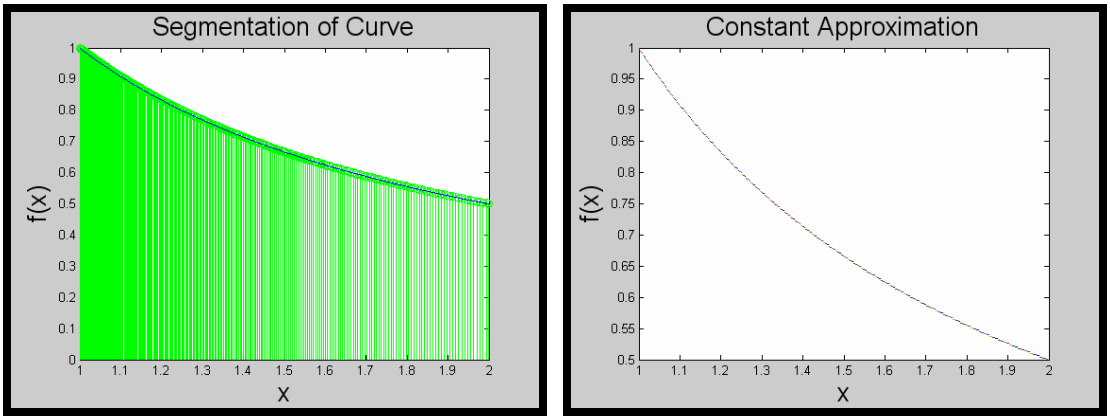


Figure A.2. Constant Approximation of $1/x$, $\varepsilon = 2^{-10}$

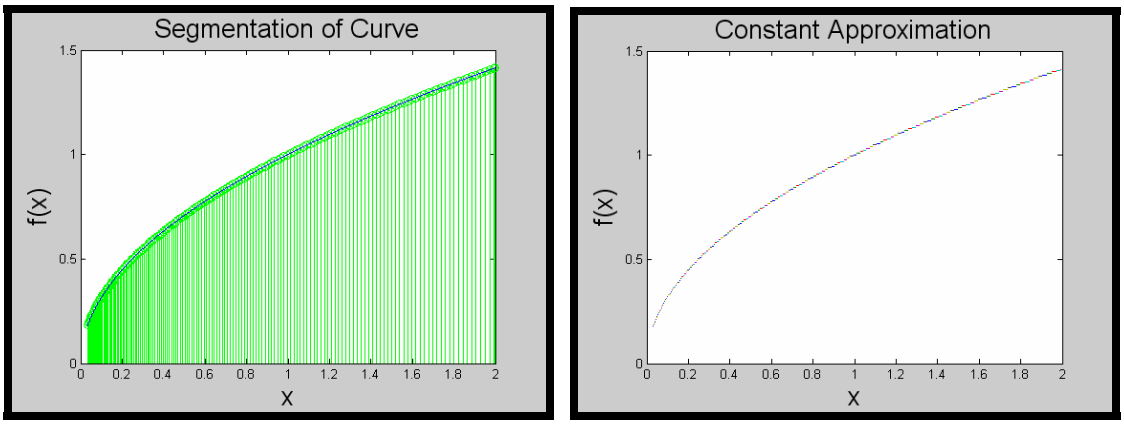


Figure A.3. Constant Approximation of $\sqrt{x}$, $\varepsilon = 2^{-8}$

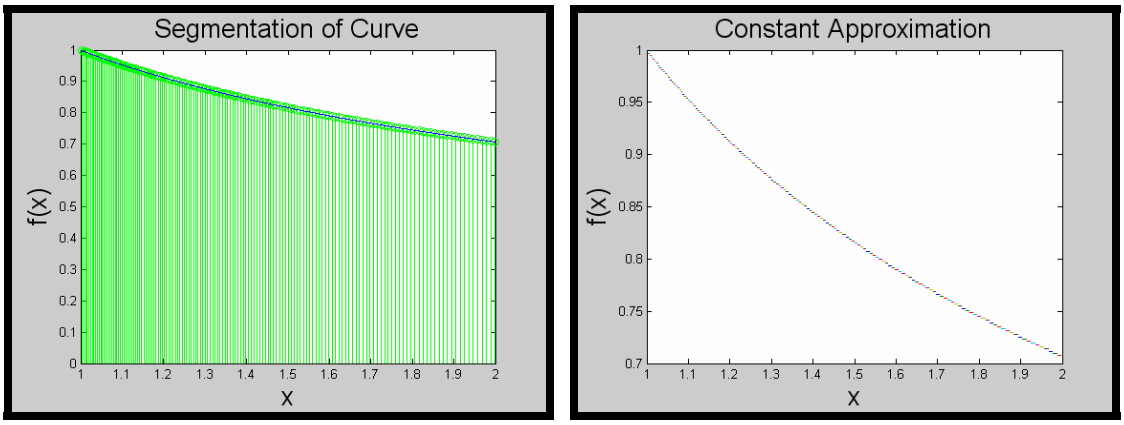


Figure A.4. Constant approximation of $1/\sqrt{x}$, $\varepsilon = 2^{-10}$

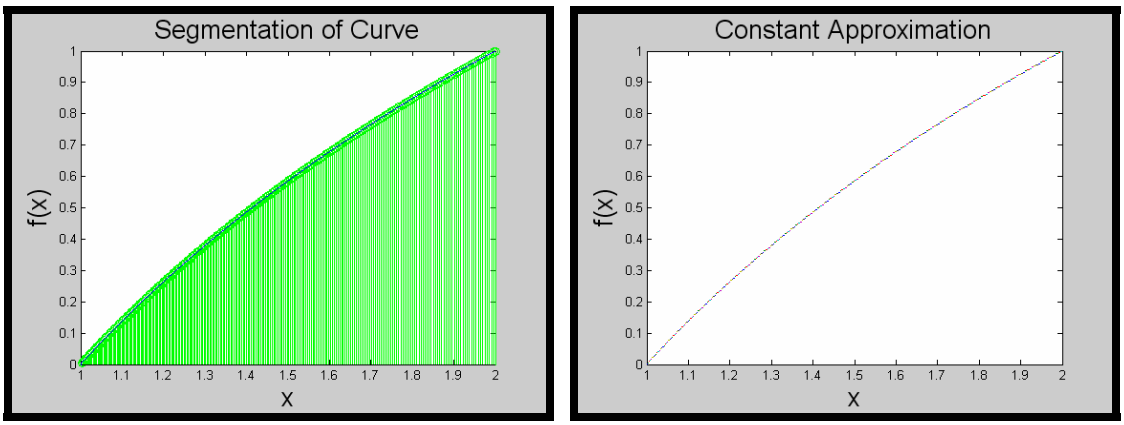


Figure A.5. Constant approximation of $\log_2(x)$, $\varepsilon = 2^{-9}$

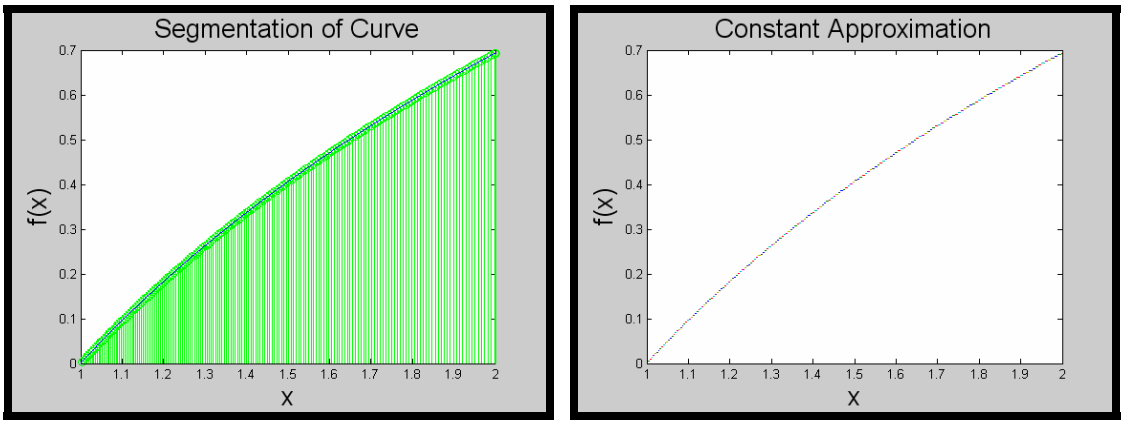


Figure A.6. Constant Approximation of $\ln(x)$, $\varepsilon = 2^{-9}$

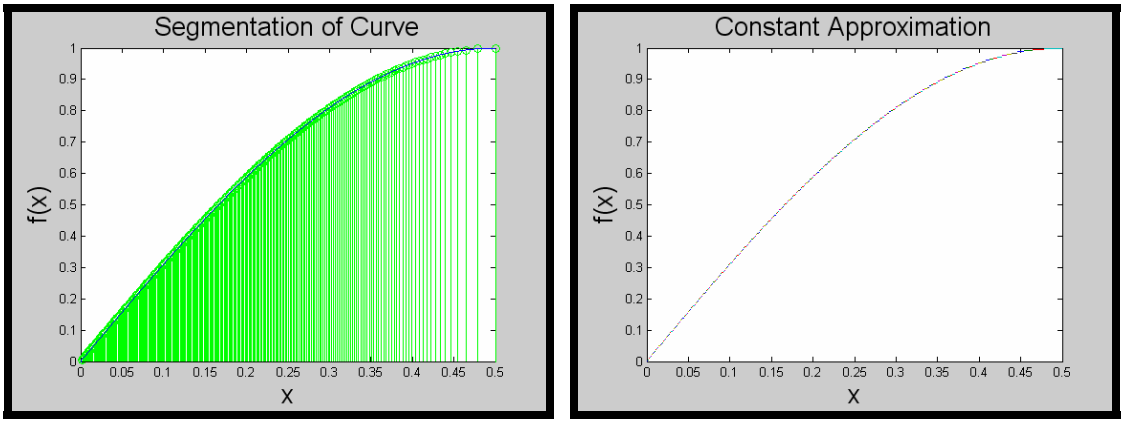


Figure A.7. Constant Approximation of $\sin(\pi x)$, $\varepsilon = 2^{-9}$

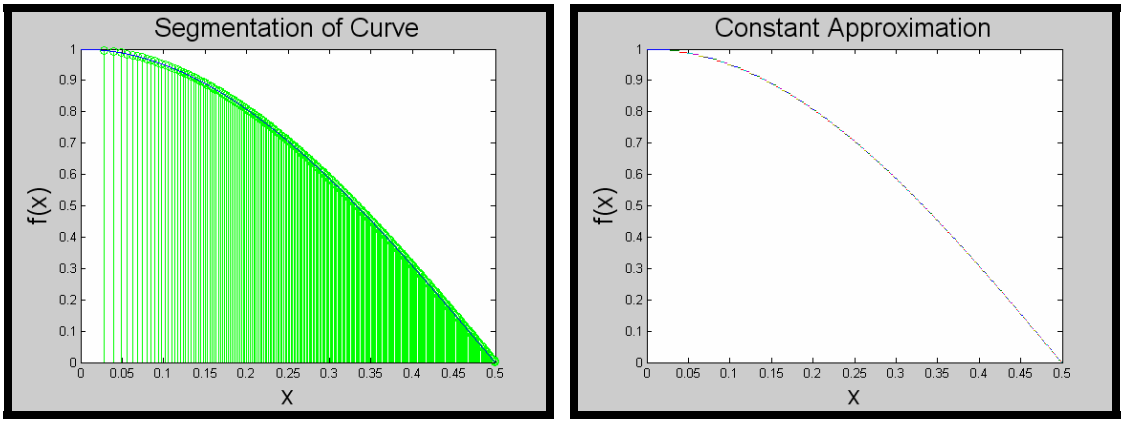


Figure A.8. Constant Approximation of $\cos(\pi x)$, $\varepsilon = 2^{-9}$

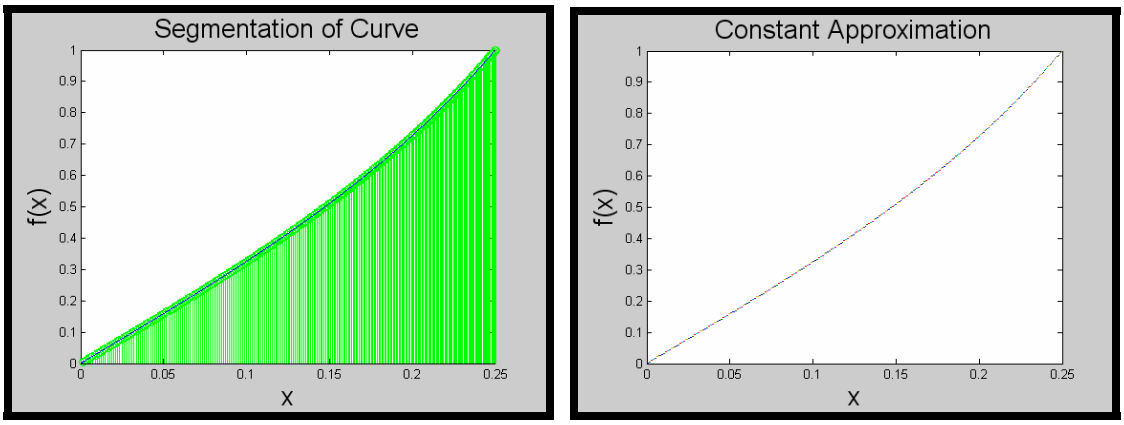


Figure A.9. Constant Approximation of $\tan(\pi x)$, $\varepsilon = 2^{-9}$

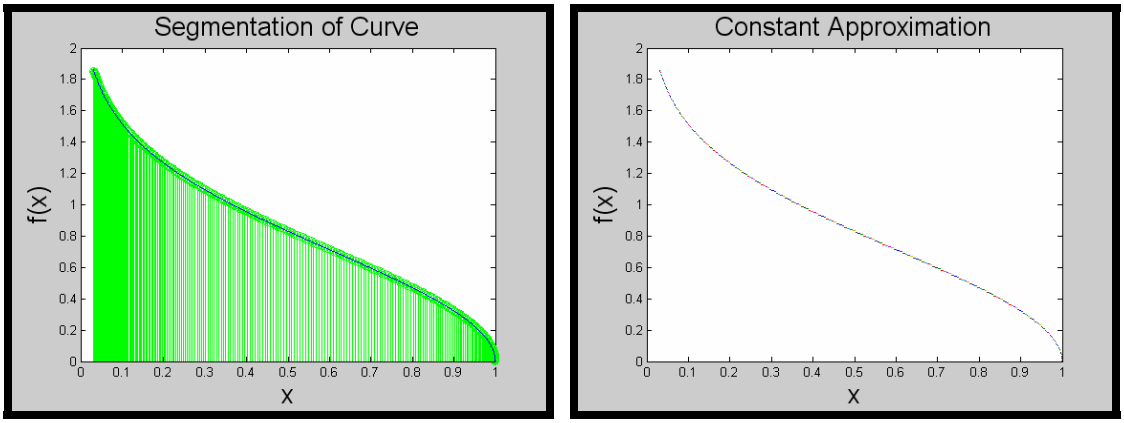


Figure A.10. Constant Approximation of $\sqrt{-\ln(x)}$, $\varepsilon = 2^{-8}$

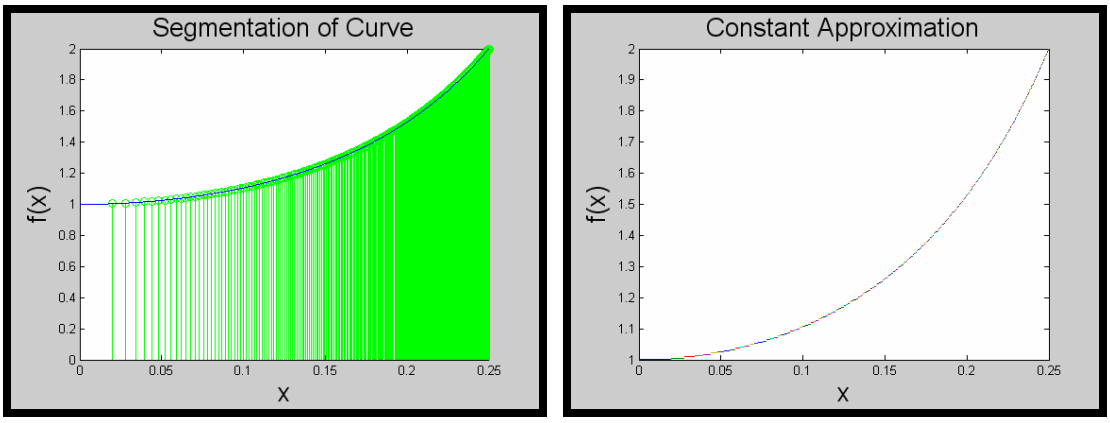


Figure A.11. Constant Approximation of $\tan^2(\pi x)+1$, $\varepsilon = 2^{-9}$

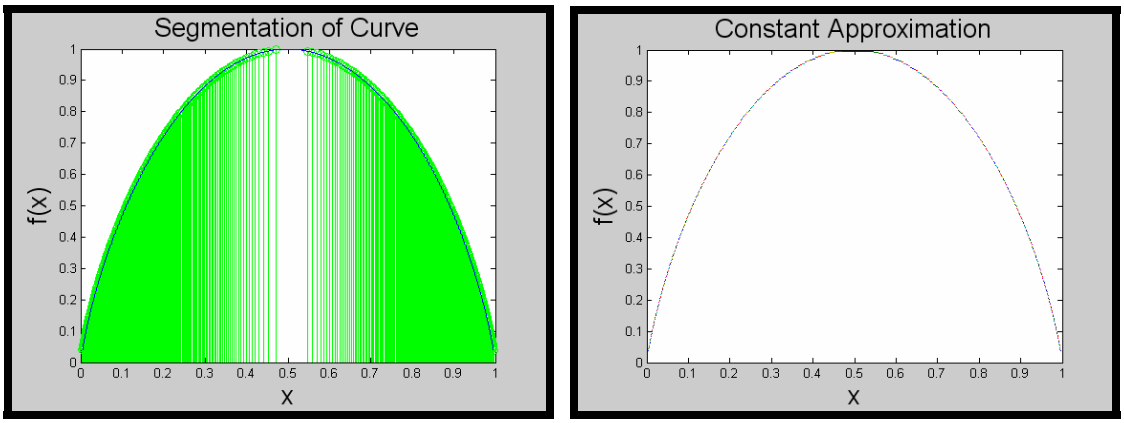


Figure A.12. Constant approximation of $-x\log_2(x)-(1-x)\log_2(1-x)$, $\varepsilon = 2^{-9}$

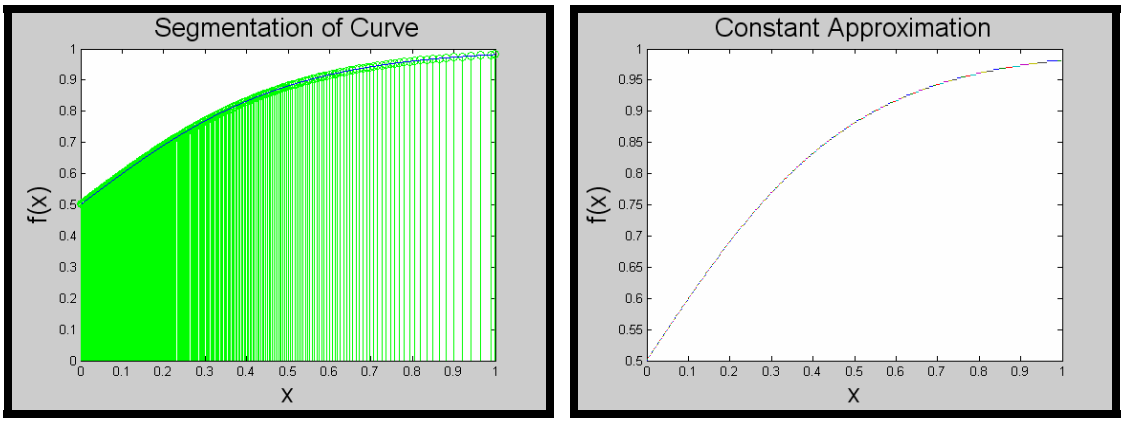


Figure A.13. Constant Approximation of $1/\left(1+e^{-4x}\right)$, $\varepsilon = 2^{-10}$

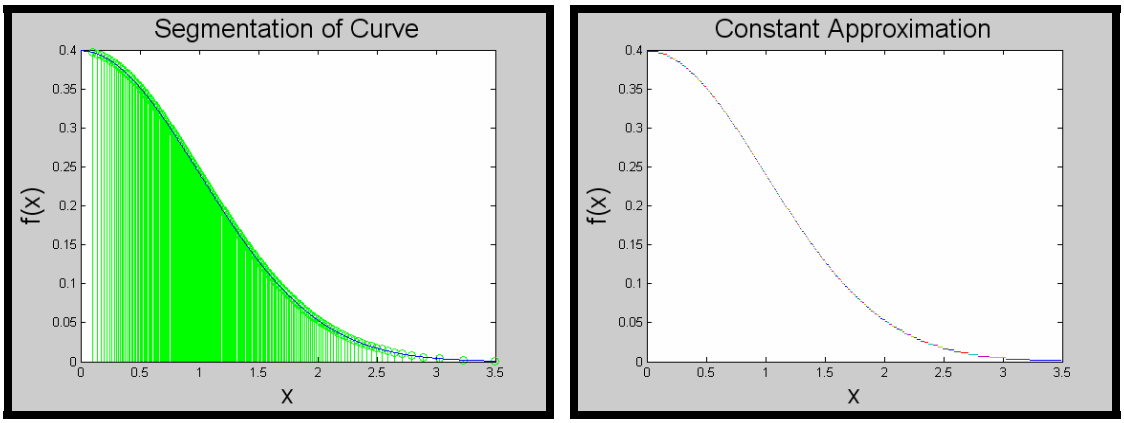


Figure A.14. Constant Approximation of $\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$, $\varepsilon = 2^{-10}$

* Domain was changed to illustrate more of the Gaussian distribution

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. POWER-OF-2-APPROXIMATION DATA

The following graphs the power-of-2 approximation of all of the functions listed in Tables 4.2 and 4.3. Note that the number of samples within the interval are 200,000..

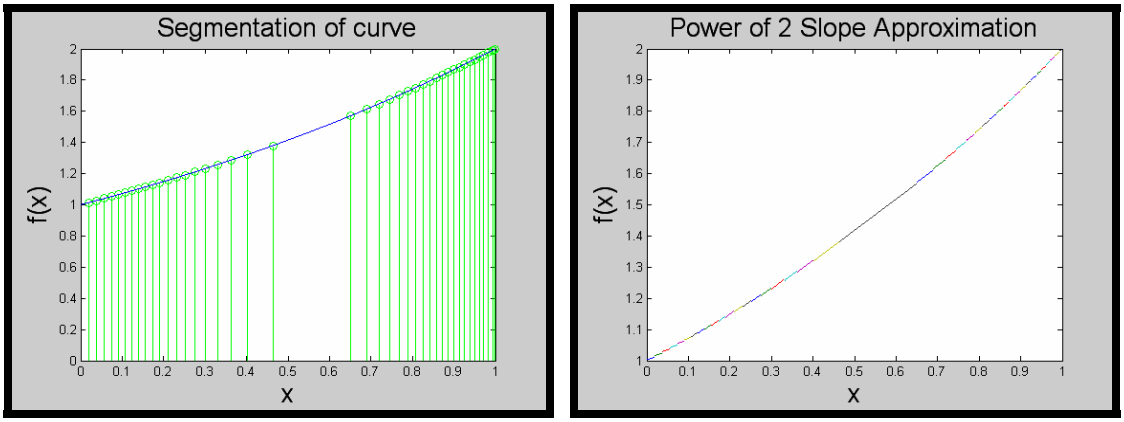
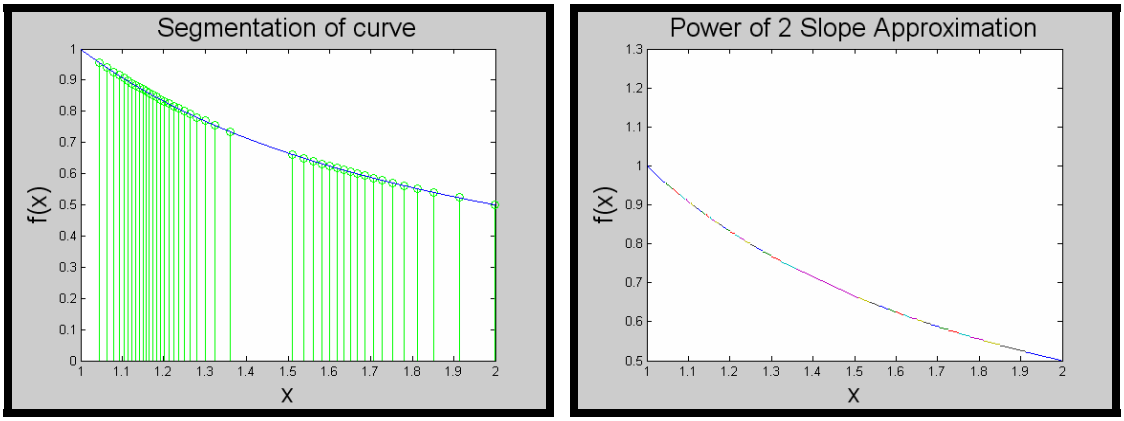


Figure B.1. Power-of-2-approximation of 2^x, $\varepsilon = 2^{-9}$



FigureB.2. Power-of-2-approximation of $1/x$, $\varepsilon = 2^{-10}$

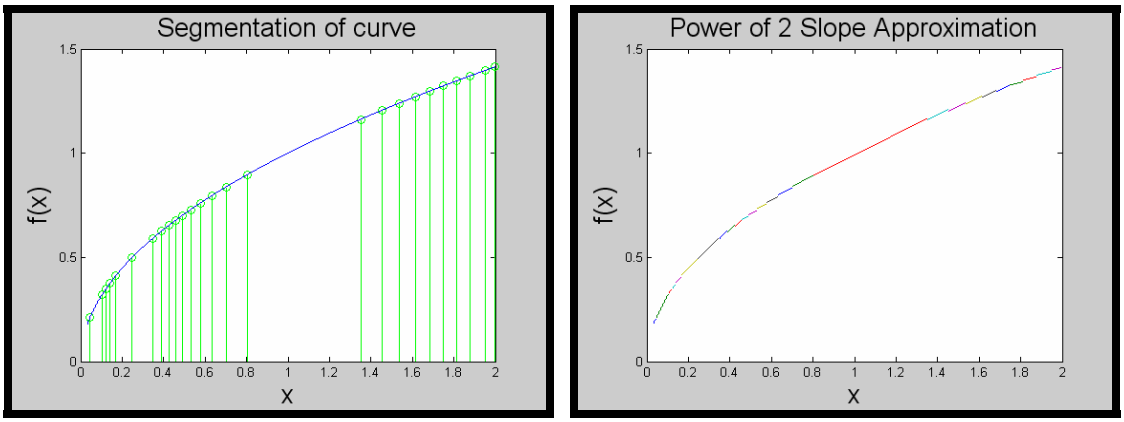


Figure B.3. Power-of-2-approximation of $\sqrt{x}$, $\varepsilon = 2^{-8}$

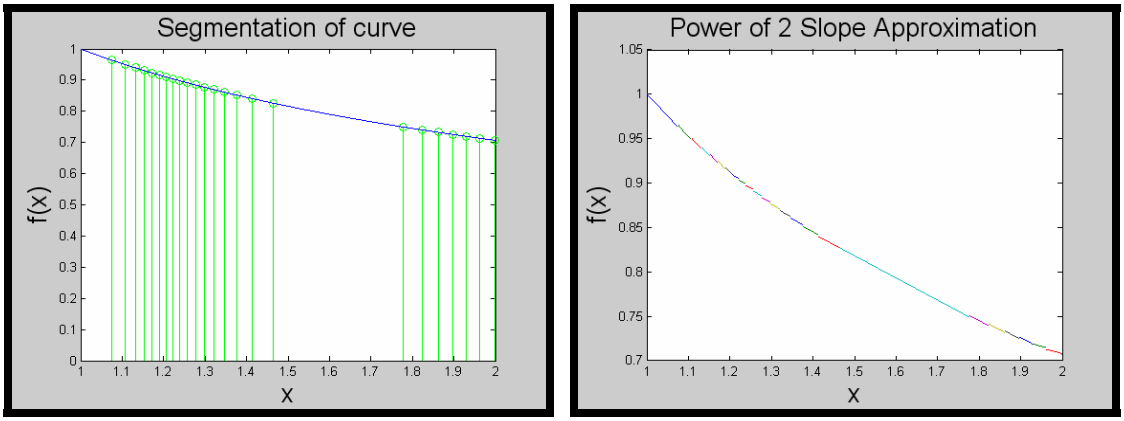
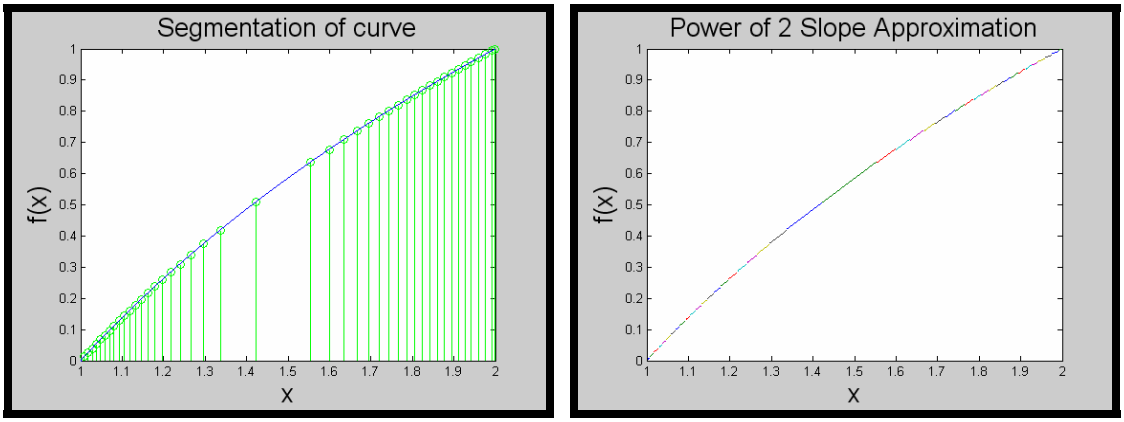
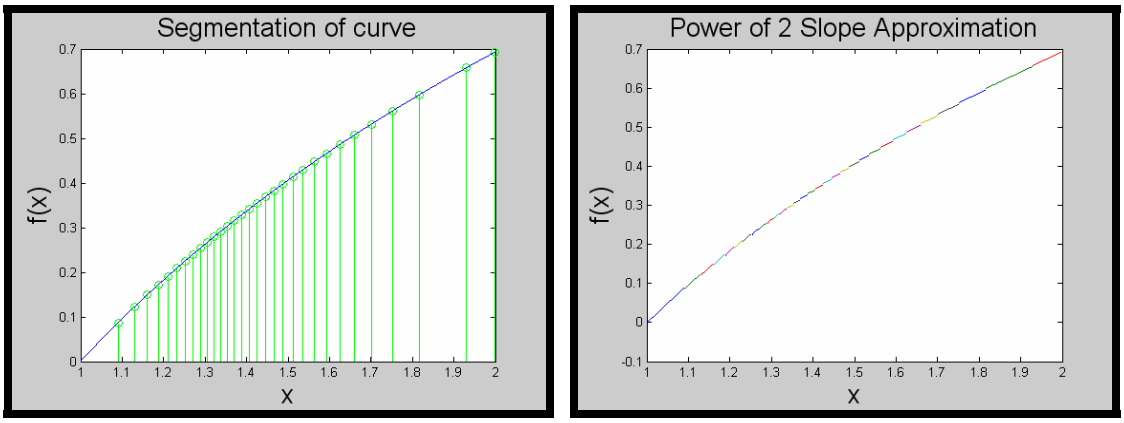


Figure B.4. Power-of-2-approximation of $1/\sqrt{x}$, $\varepsilon = 2^{-10}$



FigureB.5. Power-of-2-approximation of $\log_2(x)$, $\varepsilon = 2^{-9}$



FigureB.6. Power-of-2-approximation of $\ln(x)$, $\varepsilon = 2^{-9}$

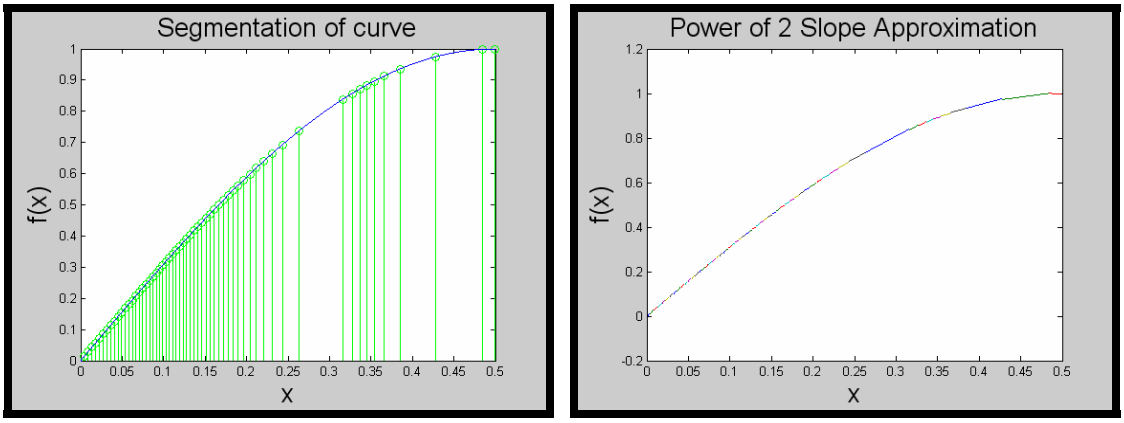


Figure B.7. Power-of-2-approximation of $\sin(\pi x)$, $\varepsilon = 2^{-9}$

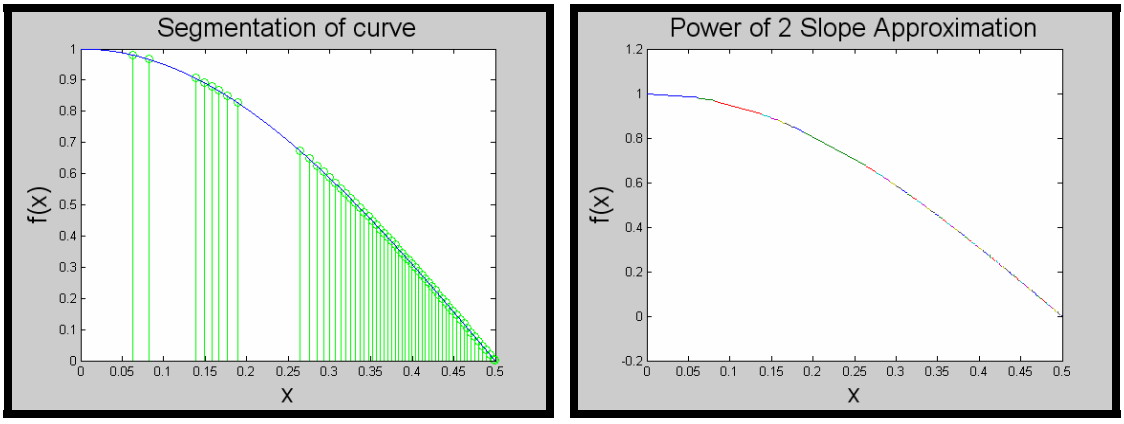
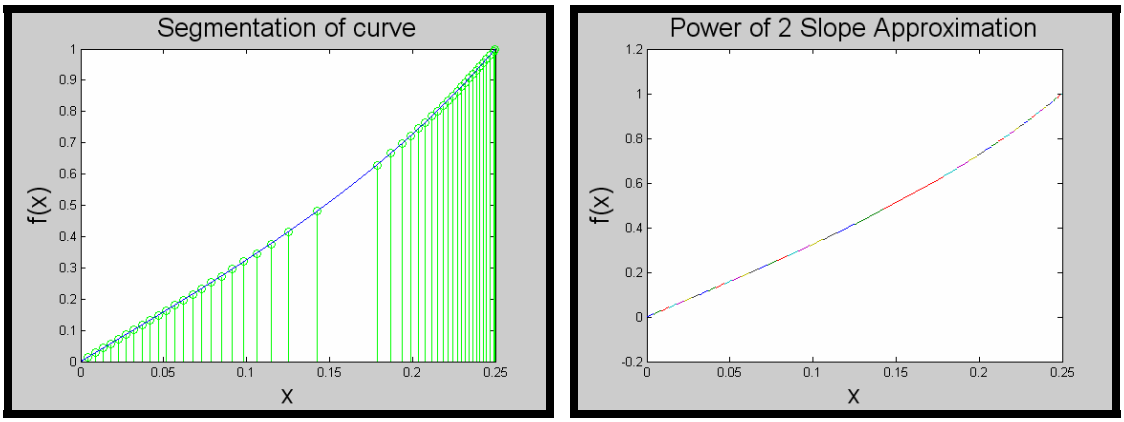


Figure B.8. Power-of-2-approximation of $\cos(\pi x)$, $\varepsilon = 2^{-9}$



FigureB.9. Power-of-2-approximation of $\tan(\pi x)$, $\varepsilon = 2^{-9}$

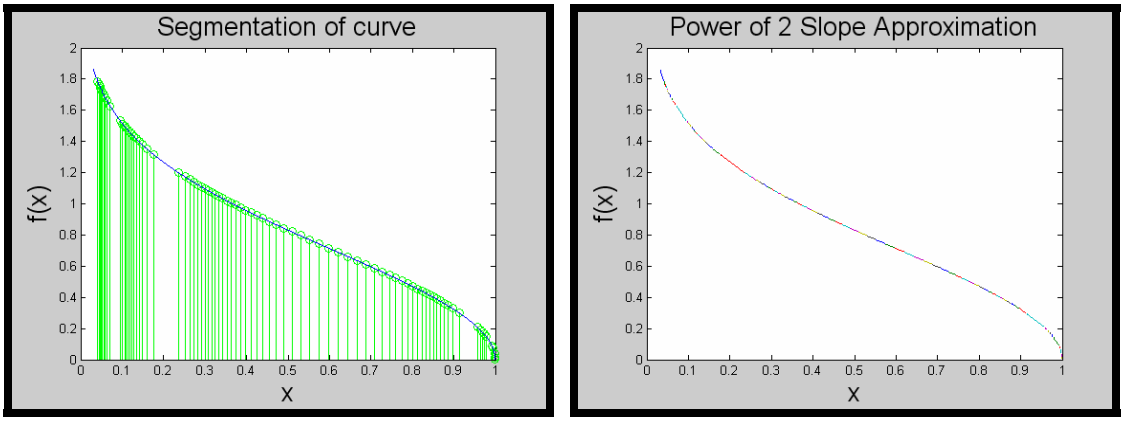


Figure B.10. Power-of-2-approximation of $\sqrt{-\ln(x)}$, $\varepsilon = 2^{-8}$

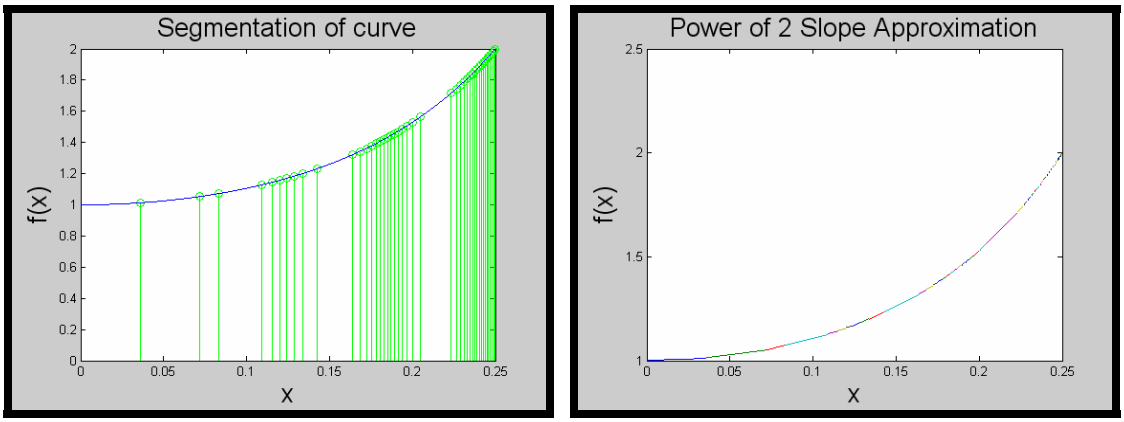


Figure B.11. Power-of-2-approximation of $\tan^2(\pi x)+1$, $\varepsilon = 2^{-9}$

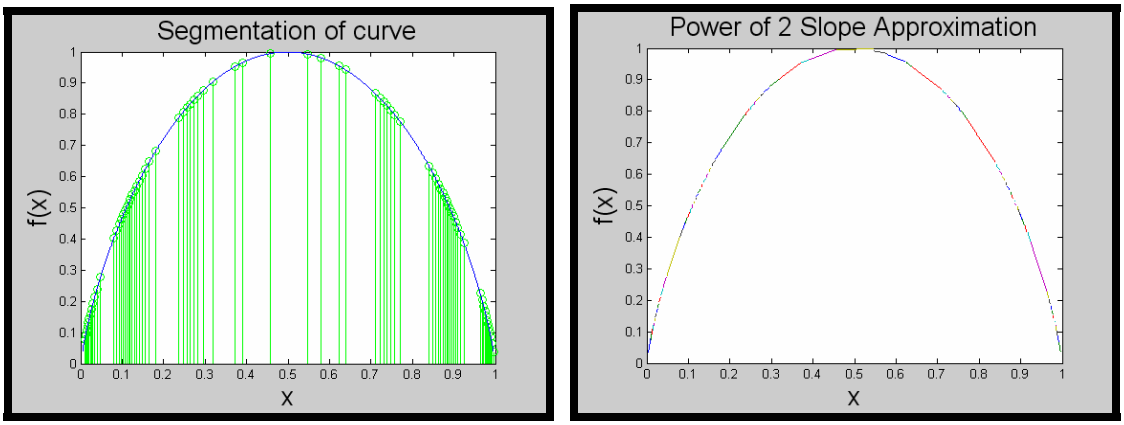


Figure B.12. Power-of-2-approximation of $-x\log_2(x)-(1-x)\log_2(1-x)$, $\varepsilon = 2^{-9}$

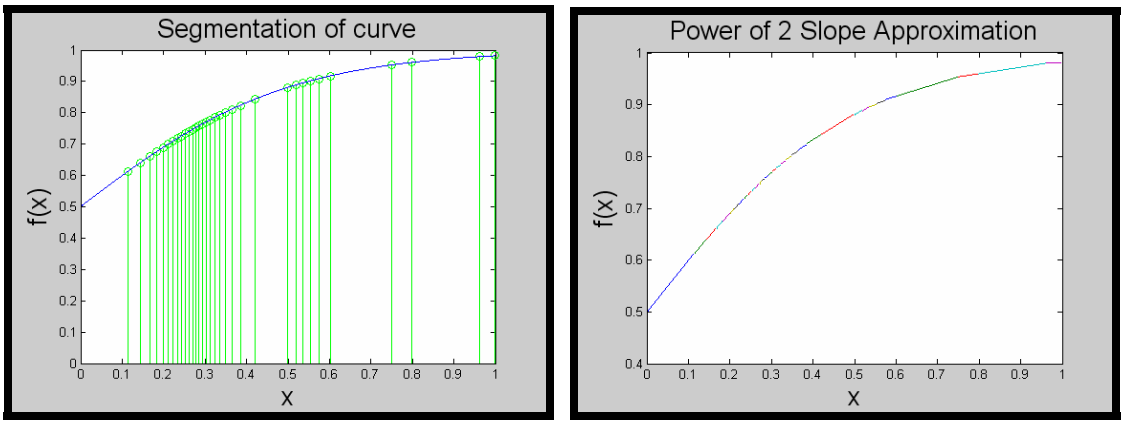


Figure B.13. Power-of-2-approximation of $1/\left(1+e^{-4x}\right)$, $\varepsilon = 2^{-10}$

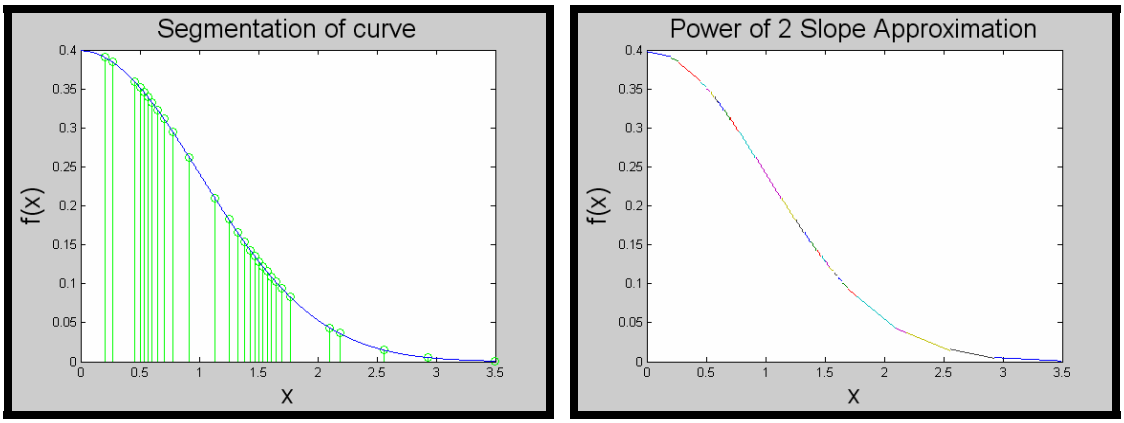


Figure B.14. Power-of-2-approximation of $\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$, $\varepsilon = 2^{-10}$

* Domain was changed to illustrate more of the Gaussian distribution

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. PIECEWISE LINEAR ALGORTIHM

The following MATLAB program calculates the number of segments needed to realize a function using the piecewise linear algorithm.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Piecewise Linear Algorithm
Developed by:  Jon T. Butler
Redesigned by:  Zaldy M. Valenzuela

Function:  Given a specified error, the program takes a function and
segments it according to that error.

Notes:  The original algorithm that was developed was not a greedy
algorithm.  Adjustments were made in order to make it a greedy
algorithm.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc
clear

x=linspace(0,.5,200000); %0:delta:1.5;
N=size(x)
N=N(2)
curve=sin(pi*x);


error=2^-1;

segment_index=1;
x_index=1;


while(x_index < N-1)
    %Segment=sin(x(x_index))
    Segment=1./((x(x_index) - 0.3).^2 + 0.01) + 1./((x(x_index) -
0.9).^2 + 0.04) - 6;

    while(max(Segment) - min(Segment) < 2*error && x_index < N)
        x_index=x_index + 1;
        Segment = [ Segment (1./((x(x_index) - 0.3).^2 + 0.01) +
1./((x(x_index) - 0.9).^2 + 0.04) - 6)];
    end

    end_segment(segment_index)=x_index-1;

    level(segment_index)= .5 * max(Segment) + .5 * min(Segment);
    segment_index=segment_index+1;

end
```

```
number_of_segments=size(end_segment);
number_of_segments=number_of_segments(2);

figure(1);
plot(x,curve);
```

APPENDIX D. CONSTANT APPROXIMATION ALGORITHM

The following MATLAB program calculates the number of segments needed to realize a function using the constant approximation algorithm.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Constant Approximation
Developed by:  Jon T. Butler and Zaldy Valenzuela
Implemented by:  Zaldy M. Valenzuela

Function:  Given a specified error, the program takes a function and
segments it according to that error using lines of slope ‘0’.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc
clear

error=2^-10;

%index variables
right_end = 1;

left_end_zero = 1;
right_end_zero = left_end_zero + 1;
max_right_end = left_end_zero + 1;
segment_edge_index_zero = 1;

overall_segment_edge_index = 1;

%defining the curve that we will compare

xmin = 1/256;
xmax = 255/256;
samples = 200000;

x=linspace(xmin,xmax,samples);
N=size(x);
N=N(2);


%curve= 1./((x - 0.3).^2 + 0.01) + 1./((x - 0.9).^2 + 0.04) - 6;
%curve = 2.^x;
%curve = 1./x;
%curve = sqrt(x);
%curve = 1./sqrt(x);
%curve = log2(x);
%curve = log(x);
%curve=sin(pi*x);
%curve = cos(pi*x);
%curve = tan(pi*x);
%curve = sqrt(-log(x));
%curve = ((tan(pi*x)).^2) + 1;
```

```
curve = (-x.*log2(x))-(1-x).*(log2(1-x));
%curve = 1./(1+exp(-4.*x));
%curve = (1./sqrt(2*pi)).*exp(-(x.^2)/2);
%curve = 1.5*x;

figure(1)
plot(x,curve)
xlabel('x','FontSize',20)
ylabel('f(x)','FontSize',20)
title ('Sine Function','FontSize',20)
%slope of 0
func_with_slope_zero = 0*x;

%subtracting the original function from the curve with slope 0

subtract_array_zero = curve - func_with_slope_zero;

%calculating segmentation

while (right_end < N)
    subtract_array_zero(left_end_zero )) < 2*error)
    while( right_end_zero < N &&
abs(subtract_array_zero(right_end_zero) - sub-
tract_array_zero(left_end_zero )) < 2*error)
        %if( subtract_array_zero(right_end_zero) >
subract_array_zero(right_end_zero -1 ) )
        right_end_zero = right_end_zero + 1;
        if( abs( subtract_array_zero(right_end_zero) > sub-
tract_array_zero(right_end_zero -1 ) ) )
        %if( abs(subtract_array_zero(right_end_zero)) >
abs(subract_array_zero(right_end_zero -1 )) )
            right_end_zero = right_end_zero +1;
        end

    end

    segment_edge_zero = right_end_zero - 1;

    overall_segment_edge(overall_segment_edge_index) = seg-
ment_edge_zero;

    pick_slope(overall_segment_edge_index) = 0;

    overall_segment_edge_index = overall_segment_edge_index + 1;

    left_end_zero = right_end_zero;
   %left_end_zero = segment_edge_zero;
    right_end_zero = left_end_zero + 1;
    right_end = right_end_zero;
    %right_end = segment_edge_zero;
end


segment_edge_vs_slope =[overall_segment_edge; pick_slope];
```

```
size_of_segment_edge = size(overall_segment_edge);
size_of_segment_edge = size_of_segment_edge(2);
element = 1;
segment_width_index = 1;

while (element < size_of_segment_edge)

   segment_width(segment_width_index) = overall_segment_edge(element +
1) - overall_segment_edge(element);
    element=element + 1;
    segment_width_index = segment_width_index + 1;

end


for (check_plot_index = 1:( overall_segment_edge_index - 1))
    check_plot(check_plot_index) =
curve(overall_segment_edge(check_plot_index));
end

    %plot(x,curve)

    %figure(2)
    %stem(x(overall_segment_edge),check_plot,'g')
    %hold all
    %plot(x,curve)
    %xlabel('x','FontSize',20)
    %ylabel('f(x)','FontSize',20)
    %title('Segmentation of Curve','FontSize',20)


   % v_cos= (abs(xmax-xmin)/samples)*overall_segment_edge;
   % index = 1;


    %while (index <= overall_segment_edge(1))
  %      new_x(index)=index;
  %      index = index + 1;
  %  end
  %  size_new_x=size(new_x);
  %  size_new_x=size_new_x(2);
  %  y=ones(1,size_new_x);
  %  y=v_cos(1)*y;
  %  figure(3)
  %  plot(new_x,y)

  size_overall_segment_edge = size(overall_segment_edge);
size_overall_segment_edge = size_overall_segment_edge(2);
test_index = 1;
index = 1;
constant_segment_index = 1;
constant_index = 1;
```

```
constant_segment=curve(1);
  while ( index <= size_overall_segment_edge)
    while ( curve(test_index) ~= curve(overall_segment_edge(index)) )
        constant_segment(constant_segment_index) = [curve(test_index)];
        test_index = test_index + 1;
        constant_segment_index = constant_segment_index + 1;

        %curve(overall_segment_edge(index))
        %test_index
    end
    y_constant(constant_index) = min(constant_segment) +
(max(constant_segment) - min(constant_segment))/2;

          %while( x_approx_index < overall_segment_edge(index))
        %    x_approx_index
        %   line_approx(line_index) = pick_slope(index)*x_approx + con-
stant(constant_index);
        %    x_approx = x_approx + 1.5000e-005;
        %    x_approx_index = x_approx_index +1;
        %    line_index = line_index + 1;

        %end
        constant_segment;
        constant_segment = 0;
    index = index + 1;
    constant_segment_index = 1;
    constant_index = constant_index + 1;
    index;
end

x_constant_index = 1;
new_segment_edge = [1 overall_segment_edge];
size_segment_edge = size(new_segment_edge);
size_segment_edge = size_segment_edge(2);

while(x_constant_index < size_segment_edge)
    x_constant(x_constant_index) =
x(new_segment_edge(x_constant_index)) +
((x(new_segment_edge(x_constant_index + 1)) - x(new_segment_edge(
x_constant_index )))/2);
    x_constant_index = x_constant_index + 1;
end

constant = y_constant - pick_slope(1,1:size_overall_segment_edge) .*
x_constant;


slope_with_constant=[pick_slope;constant;overall_segment_edge];

new_segment_edge_index = 1;

while(new_segment_edge_index <= size_overall_segment_edge)
    x_approx = lin-
space(x(new_segment_edge(new_segment_edge_index)),x(new_segment_edge(ne
```

```
w_segment_edge_index + 1)),new_segment_edge(new_segment_edge_index +
1));
    y = pick_slope(1,new_segment_edge_index).*x_approx + con-
stant(new_segment_edge_index);
        figure(3)
        title('Constant Approximation','FontSize',20)
        plot(x_approx,y)
        xlabel('x','FontSize',20)
        ylabel('f(x)','FontSize',20)
        hold all
    new_segment_edge_index = new_segment_edge_index + 1;
end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. POWER-OF-2 APROXIMATION

The following MATLAB program calculates the number of segments needed to realize a function using the power-of-2 approximation algorithm.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Power-of-2-approximation
Developed by:  Jon T. Butler and Zaldy Valenzuela
Implemented by:  Zaldy M. Valenzuela

Function:  Given a specified error, the program takes a function and
segments it according to that error using only power-of-2 slopes.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc
clear

element_num = 200000;

mult_equal_seg_index = 1;
total_segment_ambiguity = 0;
pick_slope_row = 1;
slope_error_segment = 0;
slope_error_segment_index = 1;

%Positive Number Counter
counter_1_128 = 0;
counter_1_64 = 0;
counter_1_32 = 0;
counter_1_16 = 0;
counter_1_8 = 0;
counter_one_fourth = 0;
counter_one_half = 0;
counter_1 = 0;
counter_2 = 0;
counter_4 = 0;
counter_8 = 0;
counter_16 = 0;
counter_32 = 0;
counter_64 = 0;
counter_128 = 0;

%Negative Number Counter
counter_neg_1_128 = 0;
counter_neg_1_64 = 0;
counter_neg_1_32 = 0;
counter_neg_1_16 = 0;
counter_neg_1_8 = 0;
counter_neg_one_fourth = 0;
counter_neg_one_half = 0;
counter_neg_1 = 0;
counter_neg_2 = 0;
```

```
counter_neg_4 = 0;
counter_neg_8 = 0;
counter_neg_16 = 0;
counter_neg_32 = 0;
counter_neg_64 = 0;
counter_neg_128 = 0;

%Segment Width Flag
flag = 0;

%Intializing the left end and right end for each segment
right_end = 1;

%positive numbers
left_end_1_128 = 1;
right_end_1_128 = left_end_1_128 + 1;
segment_edge_index_1_128 = 1;

left_end_1_64 = 1;
right_end_1_64 = left_end_1_64 + 1;
segment_edge_index_1_64 = 1;

left_end_1_32 = 1;
right_end_1_32 = left_end_1_32 + 1;
segment_edge_index_1_32 = 1;

left_end_1_16 = 1;
right_end_1_16 = left_end_1_16 + 1;
segment_edge_index_1_16 = 1;

left_end_1_8 = 1;
right_end_1_8 = left_end_1_8 + 1;
segment_edge_index_1_8 = 1;

left_end_one_fourth = 1;
right_end_one_fourth = left_end_one_fourth + 1;
segment_edge_index_one_fourth = 1;

left_end_one_half = 1;
right_end_one_half = left_end_one_half + 1;
segment_edge_index_one_half = 1;

left_end_one = 1;
right_end_one = left_end_one + 1;
segment_edge_index_one = 1;

left_end_two = 1;
right_end_two = left_end_two + 1;
segment_edge_index_two = 1;

left_end_4 = 1;
right_end_4 = left_end_4 + 1;
segment_edge_index_4 = 1;
```

```
left_end_8 = 1;
right_end_8 = left_end_8 + 1;
segment_edge_index_8 = 1;

left_end_16 = 1;
right_end_16 = left_end_16 + 1;
segment_edge_index_16 = 1;

left_end_32 = 1;
right_end_32 = left_end_32 + 1;
segment_edge_index_32 = 1;

left_end_64 = 1;
right_end_64 = left_end_64 + 1;
segment_edge_index_64 = 1;

left_end_128 = 1;
right_end_128 = left_end_128 + 1;
segment_edge_index_128 = 1;

%negative numbers
left_end_neg_1_128 = 1;
right_end_neg_1_128 = left_end_neg_1_128 + 1;
segment_edge_index_neg_1_128 = 1;

left_end_neg_1_64 = 1;
right_end_neg_1_64 = left_end_neg_1_64 + 1;
segment_edge_index_neg_1_64 = 1;

left_end_neg_1_32 = 1;
right_end_neg_1_32 = left_end_neg_1_32 + 1;
segment_edge_index_neg_1_32 = 1;

left_end_neg_1_16 = 1;
right_end_neg_1_16 = left_end_neg_1_16 + 1;
segment_edge_index_neg_1_16 = 1;

left_end_neg_1_8 = 1;
right_end_neg_1_8 = left_end_neg_1_8 + 1;
segment_edge_index_neg_1_8 = 1;

left_end_neg_one_fourth = 1;
right_end_neg_one_fourth = left_end_neg_one_fourth + 1;
segment_edge_index_neg_one_fourth = 1;

left_end_neg_one_half = 1;
right_end_neg_one_half = left_end_neg_one_half + 1;
segment_edge_index_neg_one_half = 1;

left_end_neg_one = 1;
right_end_neg_one = left_end_neg_one + 1;
segment_edge_index_neg_one = 1;

left_end_neg_two = 1;
```

```
right_end_neg_two = left_end_neg_two + 1;
segment_edge_index_neg_two = 1;

left_end_neg_4 = 1;
right_end_neg_4 = left_end_neg_4 + 1;
segment_edge_index_neg_4 = 1;

left_end_neg_8 = 1;
right_end_neg_8 = left_end_neg_8 + 1;
segment_edge_index_neg_8 = 1;

left_end_neg_16 = 1;
right_end_neg_16 = left_end_neg_16 + 1;
segment_edge_index_neg_16 = 1;

left_end_neg_32 = 1;
right_end_neg_32 = left_end_neg_32 + 1;
segment_edge_index_neg_32 = 1;

left_end_neg_64 = 1;
right_end_neg_64 = left_end_neg_64 + 1;
segment_edge_index_neg_64 = 1;

left_end_neg_128 = 1;
right_end_neg_128 = left_end_neg_128 + 1;
segment_edge_index_neg_128 = 1;

overall_segment_edge_index = 1;
slope_index = 1;

    x_approx = 0;
    x_approx_index = 1;
    line_index = 1;
%segment counter for slopes with equal widths
flag_counter_neg_128 = 0;
flag_counter_neg_64 = 0;
flag_counter_neg_32= 0;
flag_counter_neg_16 = 0;
flag_counter_neg_8 = 0;
flag_counter_neg_4 = 0;
flag_counter_neg_2 = 0;
flag_counter_neg_1 = 0;
flag_counter_neg_1_2 = 0;
flag_counter_neg_1_4 = 0;
flag_counter_neg_1_8 = 0;
flag_counter_neg_1_16 = 0;
flag_counter_neg_1_32 = 0;
flag_counter_neg_1_64 = 0;
flag_counter_neg_1_128 = 0;
flag_counter_1_128 = 0;
flag_counter_1_64 = 0;
flag_counter_1_32 = 0;
flag_counter_1_16 = 0;
flag_counter_1_8 = 0;
flag_counter_1_4 = 0;
```

```
flag_counter_1_2 = 0;
flag_counter_1 = 0;
flag_counter_2 = 0;
flag_counter_4 = 0;
flag_counter_8 = 0;
flag_counter_16 = 0;
flag_counter_32 = 0;
flag_counter_64 = 0;
flag_counter_128 =0;
%defining the curve that we will compare

error = 2^-8;
x=linspace(1/32,2,element_num);
N=size(x);
N=N(2);
%curve = (1./((x - 0.3).^2 + 0.01) + 1./((x - 0.9).^2 + 0.04) - 6);
%curve=(sqrt(-log(x)));
%curve=(x.*log2(x)-(1-x).*(log2(1-x)));
%curve=(1/sqrt(2*pi))*exp(-(x.^2)/2);
curve = sqrt(x);
%slope of 1/128
%curve = sin(x);
%curve= 1./((x - 0.3).^2 + 0.01) + 1./((x - 0.9).^2 + 0.04) - 6;
%curve = 2.^x;
%curve = 1./x;
%curve = sqrt(x);
%curve = 1./sqrt(x);
%curve = log2(x);
%curve = log(x);
%curve=sin(pi*x);
%curve = cos(pi*x);
%curve = tan(pi*x);
%curve = sqrt(-log(x));
%curve = ((tan(pi*x)).^2) + 1;
%curve = (-x.*log2(x))-(1-x).*(log2(1-x));
%curve = 1./(1+exp(-4.*x));
%curve = (1./sqrt(2*pi)).*exp(-(x.^2)/2);
%curve = 1.5*x;

func_with_slope_1_128 = (1/128)*x;

%slope of 1/64
func_with_slope_1_64 = (1/64)*x;

%slope of 1/32
func_with_slope_1_32 = (1/32)*x;

%slope of 1/16
func_with_slope_1_16 = (1/16)*x;

%slope of 1/8
func_with_slope_1_8 = (1/8)*x;

%slope of 1/4
func_with_slope_one_fourth = (1/4)*x;
```

```
%slope of 1/2
func_with_slope_one_half = .5*x;

%slope of 1
func_with_slope_one = 1*x;

%slope of 2
func_with_slope_two = 2*x;

%slope of 4
func_with_slope_4 = 4*x;

%slope of 8
func_with_slope_8 = 8*x;

%slope of 16
func_with_slope_16 = 16*x;

%slope of 32
func_with_slope_32 = 32*x;

%slope of 64
func_with_slope_64 = 64*x;

%slope of 128
func_with_slope_128 = 128*x;

%slope of -1/128
func_with_slope_neg_1_128 = (-1/128)*x;

%slope of -1/64
func_with_slope_neg_1_64 = (-1/64)*x;

%slope of -1/32
func_with_slope_neg_1_32 = (-1/32)*x;

%slope of -1/16
func_with_slope_neg_1_16 = (-1/16)*x;

%slope of -1/8
func_with_slope_neg_1_8 = (-1/8)*x;

%slope of -1/4
func_with_slope_neg_one_fourth = -.25*x;

%slope of -1/2
func_with_slope_neg_one_half = -.5*x;

%slope of -1
func_with_slope_neg_one = -1*x;

%slope of -2
```

```
func_with_slope_neg_two = -2*x;

%slope of -4
func_with_slope_neg_4 = -4*x;

%slope of -8
func_with_slope_neg_8 = -8*x;

%slope of -16
func_with_slope_neg_16 = -16*x;

%slope of -32
func_with_slope_neg_32 = -32*x;

%slope of -64
func_with_slope_neg_64 = -64*x;

%slope of -128
func_with_slope_neg_128 = -128*x;

segment_count_array = [-(2^7) -(2^6) -(2^5) -(2^4) -(2^3) -(2^2) -(2^1)
-(2^0) -(2^(-1)) -(2^(-2)) -(2^(-3)) -(2^(-4)) -(2^(-5)) -(2^(-6)) -
(2^(-7)) 2^(-7) 2^(-6) 2^(-5) 2^(-4) 2^(-3) 2^(-2) 2^(-1) 2^0 2^1 2^2
2^3 2^4 2^5 2^6 2^7];

%subtracting the original functions from the curves
%subtract_array_zero = curve - func_with_slope_zero;

subtract_array_1_128 = curve - func_with_slope_1_128;
subtract_array_1_64 = curve - func_with_slope_1_64;
subtract_array_1_32 = curve - func_with_slope_1_32;
subtract_array_1_16 = curve - func_with_slope_1_16;
subtract_array_1_8 = curve - func_with_slope_1_8;
subtract_array_one_fourth = curve - func_with_slope_one_fourth;
subtract_array_one_half = curve - func_with_slope_one_half;
subtract_array_one = curve - func_with_slope_one;
subtract_array_two = curve - func_with_slope_two;
subtract_array_4 = curve - func_with_slope_4;
subtract_array_8 = curve - func_with_slope_8;
subtract_array_16 = curve - func_with_slope_16;
subtract_array_32 = curve - func_with_slope_32;
subtract_array_64 = curve - func_with_slope_64;
subtract_array_128 = curve - func_with_slope_128;

subtract_array_neg_1_128 = curve - func_with_slope_neg_1_128;
subtract_array_neg_1_64 = curve - func_with_slope_neg_1_64;
subtract_array_neg_1_32 = curve - func_with_slope_neg_1_32;
subtract_array_neg_1_16 = curve - func_with_slope_neg_1_16;
subtract_array_neg_1_8 = curve - func_with_slope_neg_1_8;
subtract_array_neg_one_fourth = curve - func_with_slope_neg_one_fourth;
subtract_array_neg_one_half = curve - func_with_slope_neg_one_half;
subtract_array_neg_one = curve - func_with_slope_neg_one;
subtract_array_neg_two = curve - func_with_slope_neg_two;
subtract_array_neg_4 = curve - func_with_slope_neg_4;
subtract_array_neg_8 = curve - func_with_slope_neg_8;
```

```
subtract_array_neg_16 = curve - func_with_slope_neg_16;
subtract_array_neg_32 = curve - func_with_slope_neg_32;
subtract_array_neg_64 = curve - func_with_slope_neg_64;
subtract_array_neg_128 = curve - func_with_slope_neg_128;


plot(x,subtract_array_one);
%calculating segmentation

while (right_end < element_num  - 1)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                    positive slopes
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %calculating size of segments for function of slope of 1/8
    while(right_end_1_8 < N && abs(subtract_array_1_8(right_end_1_8) -
subtract_array_1_8(left_end_1_8 )) < 2*error)
        right_end_1_8 = right_end_1_8 + 1;
    end
    segment_edge_1_8 = right_end_1_8 - 1;

    %calculating size of segments for function of slope of 1/16
    while(right_end_1_16 < N && abs(subtract_array_1_16(right_end_1_16)
- subtract_array_1_16(left_end_1_16 )) < 2*error)
        right_end_1_16 = right_end_1_16 + 1;
    end
    segment_edge_1_16 = right_end_1_16 - 1;

    %calculating size of segments for function of slope of 1/32
    while(right_end_1_32 < N && abs(subtract_array_1_32(right_end_1_32)
- subtract_array_1_32(left_end_1_32 )) < 2*error)
        right_end_1_32 = right_end_1_32 + 1;
    end
    segment_edge_1_32 = right_end_1_32 - 1;

    %calculating size of segments for function of slope of 1/64
    while(right_end_1_64 < N && abs(subtract_array_1_64(right_end_1_64)
- subtract_array_1_64(left_end_1_64 )) < 2*error)
        right_end_1_64 = right_end_1_64 + 1;
    end
    segment_edge_1_64 = right_end_1_64 - 1;

    %calculating size of segments for function of slope of 1/128
    while(right_end_1_128 < N &&
abs(subtract_array_1_128(right_end_1_128) - sub-
tract_array_1_128(left_end_1_128 )) < 2*error)
        right_end_1_128 = right_end_1_128 + 1;
    end
    segment_edge_1_128 = right_end_1_128 - 1;

    %calculating size of segments for function of slope 1/4
```

```
    while(right_end_one_fourth < N &&
abs(subtract_array_one_fourth(right_end_one_fourth) - sub-
tract_array_one_fourth(left_end_one_fourth )) < 2*error)
        right_end_one_fourth = right_end_one_fourth + 1;
    end
    segment_edge_one_fourth = right_end_one_fourth - 1;

    %calculating size of segments for function of slope 1/2
    while(right_end_one_half < N &&
abs(subtract_array_one_half(right_end_one_half) - sub-
tract_array_one_half(left_end_one_half )) < 2*error)
        right_end_one_half = right_end_one_half + 1;
    end
    segment_edge_one_half = right_end_one_half - 1;

    %calculating size of segments for function of slope of 1
    while(right_end_one < N && abs(subtract_array_one(right_end_one) -
subtract_array_one(left_end_one )) < 2*error)
        right_end_one = right_end_one + 1;
    end
    segment_edge_one = right_end_one - 1;

    %calculating size of segments for function of slope of 2
    while(right_end_two < N && abs(subtract_array_two(right_end_two) -
subtract_array_two(left_end_two )) < 2*error)
        right_end_two = right_end_two + 1;
    end
    segment_edge_two = right_end_two - 1;

    %calculating size of segments for function of slope of 4
    while(right_end_4 < N && abs(subtract_array_4(right_end_4) - sub-
tract_array_4(left_end_4 )) < 2*error)
        right_end_4 = right_end_4 + 1;
    end
    segment_edge_4 = right_end_4 - 1;

    %calculating size of segments for function of slope of 8
    while(right_end_8 < N && abs(subtract_array_8(right_end_8) - sub-
tract_array_8(left_end_8 )) < 2*error)
        right_end_8 = right_end_8 + 1;
    end
    segment_edge_8 = right_end_8 - 1;

    %calculating size of segments for function of slope of 16
    while(right_end_16 < N && abs(subtract_array_16(right_end_16) -
subtract_array_16(left_end_16 )) < 2*error)
        right_end_16 = right_end_16 + 1;
    end
    segment_edge_16 = right_end_16 - 1;

    %calculating size of segments for function of slope of 32
    while(right_end_32 < N && abs(subtract_array_32(right_end_32) -
subtract_array_32(left_end_32 )) < 2*error)
        right_end_32 = right_end_32 + 1;
    end
```

```
    segment_edge_32 = right_end_32 - 1;

    %calculating size of segments for function of slope of 64
    while(right_end_64 < N && abs(subtract_array_64(right_end_64) -
subtract_array_64(left_end_64 )) < 2*error)
        right_end_64 = right_end_64 + 1;
    end
    segment_edge_64 = right_end_64 - 1;

    %calculating size of segments for function of slope of 128
    while(right_end_128 < N && abs(subtract_array_128(right_end_128) -
subtract_array_128(left_end_128 )) < 2*error)
        right_end_128 = right_end_128 + 1;
    end
    segment_edge_128 = right_end_128 - 1;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                    negative slopes
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%calculating size of segments for function of slope of -1/8
    while(right_end_neg_1_8 < N &&
abs(subtract_array_neg_1_8(right_end_neg_1_8) - sub-
tract_array_neg_1_8(left_end_neg_1_8 )) < 2*error)
        right_end_neg_1_8 = right_end_neg_1_8 + 1;
    end
    segment_edge_neg_1_8 = right_end_neg_1_8 - 1;

    %calculating size of segments for function of slope of -1/16
    while(right_end_neg_1_16 < N &&
abs(subtract_array_neg_1_16(right_end_neg_1_16) - sub-
tract_array_neg_1_16(left_end_neg_1_16 )) < 2*error)
        right_end_neg_1_16 = right_end_neg_1_16 + 1;
    end
    segment_edge_neg_1_16 = right_end_neg_1_16 - 1;

    %calculating size of segments for function of slope of -1/32
    while(right_end_neg_1_32 < N &&
abs(subtract_array_neg_1_32(right_end_neg_1_32) - sub-
tract_array_neg_1_32(left_end_neg_1_32 )) < 2*error)
        right_end_neg_1_32 = right_end_neg_1_32 + 1;
    end
    segment_edge_neg_1_32 = right_end_neg_1_32 - 1;

    %calculating size of segments for function of slope of -1/64
    while(right_end_neg_1_64 < N &&
abs(subtract_array_neg_1_64(right_end_neg_1_64) - sub-
tract_array_neg_1_64(left_end_neg_1_64 )) < 2*error)
        right_end_neg_1_64 = right_end_neg_1_64 + 1;
    end
    segment_edge_neg_1_64 = right_end_neg_1_64 - 1;

    %calculating size of segments for function of slope of -1/128
```

```
    while(right_end_neg_1_128 < N &&
abs(subtract_array_neg_1_128(right_end_neg_1_128) - sub-
tract_array_neg_1_128(left_end_neg_1_128 )) < 2*error)
        right_end_neg_1_128 = right_end_neg_1_128 + 1;
    end
    segment_edge_neg_1_128 = right_end_neg_1_128 - 1;

    %calculating size of segments for function of slope -1/4
    while(right_end_neg_one_fourth < N &&
abs(subtract_array_neg_one_fourth(right_end_neg_one_fourth) - sub-
tract_array_neg_one_fourth(left_end_neg_one_fourth )) < 2*error)
        right_end_neg_one_fourth = right_end_neg_one_fourth + 1;
    end
    segment_edge_neg_one_fourth = right_end_neg_one_fourth - 1;

    %calculating size of segments for function of slope -1/2
    while(right_end_neg_one_half < N &&
abs(subtract_array_neg_one_half(right_end_neg_one_half) - sub-
tract_array_neg_one_half(left_end_neg_one_half )) < 2*error)
        right_end_neg_one_half = right_end_neg_one_half + 1;
    end
    segment_edge_neg_one_half = right_end_neg_one_half - 1;

    %calculating size of segments for a function of slope of -1
    while(right_end_neg_one < N &&
abs(subtract_array_neg_one(right_end_neg_one) - sub-
tract_array_neg_one(left_end_neg_one )) < 2*error)
        right_end_neg_one = right_end_neg_one + 1;
    end
    segment_edge_neg_one = right_end_neg_one - 1;

    %calculating size of segments for function of slope of -2
    while(right_end_neg_two < N &&
abs(subtract_array_neg_two(right_end_neg_two) - sub-
tract_array_neg_two(left_end_neg_two )) < 2*error)
        right_end_neg_two = right_end_neg_two + 1;
    end
    segment_edge_neg_two = right_end_neg_two - 1;

    %calculating size of segments for function of slope of -4
    while(right_end_neg_4 < N &&
abs(subtract_array_neg_4(right_end_neg_4) - sub-
tract_array_neg_4(left_end_neg_4 )) < 2*error)
        right_end_neg_4 = right_end_neg_4 + 1;
    end
    segment_edge_neg_4 = right_end_neg_4 - 1;

    %calculating size of segments for function of slope of -8
    while(right_end_neg_8 < N &&
abs(subtract_array_neg_8(right_end_neg_8) - sub-
tract_array_neg_8(left_end_neg_8 )) < 2*error)
        right_end_neg_8 = right_end_neg_8 + 1;
    end
    segment_edge_neg_8 = right_end_neg_8 - 1;
```

```
    %calculating size of segments for function of slope of -16
    while(right_end_neg_16 < N &&
abs(subtract_array_neg_16(right_end_neg_16) - sub-
tract_array_neg_16(left_end_neg_16 )) < 2*error)
        right_end_neg_16 = right_end_neg_16 + 1;
    end
    segment_edge_neg_16 = right_end_neg_16 - 1;

    %calculating size of segments for function of slope of -32
    while(right_end_neg_32 < N &&
abs(subtract_array_neg_32(right_end_neg_32) - sub-
tract_array_neg_32(left_end_neg_32 )) < 2*error)
        right_end_neg_32 = right_end_neg_32 + 1;
    end
    segment_edge_neg_32 = right_end_neg_32 - 1;

    %calculating size of segments for function of slope of -64
    while(right_end_neg_64 < N &&
abs(subtract_array_neg_64(right_end_neg_64) - sub-
tract_array_neg_64(left_end_neg_64 )) < 2*error)
        right_end_neg_64 = right_end_neg_64 + 1;
    end
    segment_edge_neg_64 = right_end_neg_64 - 1;

    %calculating size of segments for function of slope of -128
    while(right_end_neg_128 < N &&
abs(subtract_array_neg_128(right_end_neg_128) - sub-
tract_array_neg_128(left_end_neg_128 )) < 2*error)
        right_end_neg_128 = right_end_neg_128 + 1;
    end
    segment_edge_neg_128 = right_end_neg_128 - 1;


    %figuring out which segment is the largest and resetting the seg-
ment
    %lenth

    all_right_edge = [ segment_edge_neg_128 segment_edge_neg_64 seg-
ment_edge_neg_32 segment_edge_neg_16 segment_edge_neg_8 seg-
ment_edge_neg_4 segment_edge_neg_two segment_edge_neg_one seg-
ment_edge_neg_one_half segment_edge_neg_one_fourth segment_edge_neg_1_8
segment_edge_neg_1_16 segment_edge_neg_1_32 segment_edge_neg_1_64 seg-
ment_edge_neg_1_128 segment_edge_1_128 segment_edge_1_64 seg-
ment_edge_1_32 segment_edge_1_16 segment_edge_1_8 seg-
ment_edge_one_fourth segment_edge_one_half segment_edge_one seg-
ment_edge_two segment_edge_4 segment_edge_8 segment_edge_16 seg-
ment_edge_32 segment_edge_64 segment_edge_128 ];
    overall_segment_edge(overall_segment_edge_index) =
max(all_right_edge);

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_1_128 )
        pick_slope(1,slope_index) = 1/128;
        counter_1_128 = counter_1_128 + 1;
        flag = flag + 1;
```

```
        flag_counter_1_128 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_1_64 )
        pick_slope(1,slope_index) = 1/64;
        counter_1_64 = counter_1_64 + 1;
        flag = flag + 1;
        flag_counter_1_64 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_1_32 )
        pick_slope(1,slope_index) = 1/32;
        counter_1_32 = counter_1_32 + 1;
        flag = flag + 1;
        flag_counter_1_32 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_1_16 )
        pick_slope(1,slope_index) = 1/16;
        counter_1_16 = counter_1_16 + 1;
        flag = flag + 1;
        flag_counter_1_16 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_1_8 )
        pick_slope(1,slope_index) = 1/8;
        counter_1_8 = counter_1_8 + 1;
        flag = flag + 1;
        flag_counter_1_8 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_one_fourth )
        pick_slope(1,slope_index) = 1/4;
        counter_one_fourth = counter_one_fourth + 1;
        flag = flag + 1;
        flag_counter_1_4 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_one_half )
        pick_slope(1,slope_index) = 1/2;
        counter_one_half = counter_one_half + 1;
        flag = flag + 1;
        flag_counter_1_2 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_one )
        pick_slope(1,slope_index) = 1;
        counter_1 = counter_1 + 1;
```

```
        flag = flag + 1;
        flag_counter_1 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_two )
        pick_slope(1,slope_index) = 2;
        counter_2 = counter_2 + 1;
        flag = flag + 1;
        flag_counter_2 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_4 )
        pick_slope(1,slope_index) = 4;
        counter_4 = counter_4 + 1;
        flag = flag + 1;
        flag_counter_4 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_8 )
        pick_slope(1,slope_index) = 8;
        counter_8 = counter_8 + 1;
        flag = flag + 1;
        flag_counter_8 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_16 )
        pick_slope(1,slope_index) = 16;
        counter_16 = counter_16 + 1;
        flag = flag + 1;
        flag_counter_16 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_32 )
        pick_slope(1,slope_index) = 32;
        counter_32 = counter_32 + 1;
        flag = flag + 1;
        flag_counter_32 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_64 )
        pick_slope(1,slope_index) = 64;
        counter_64 = counter_64 + 1;
        flag = flag + 1;
        flag_counter_64 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_128 )
        pick_slope(1,slope_index) = 128;
```

```
        counter_128 = counter_128 + 1;
        flag = flag + 1;
        flag_counter_128 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_1_128 )
        pick_slope(1,slope_index) = -1/128;
        counter_neg_1_128 = counter_neg_1_128 + 1;
        flag = flag + 1;
        flag_counter_neg_1_128 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_1_64 )
        pick_slope(1,slope_index) = -1/64;
        counter_neg_1_64 = counter_neg_1_64 + 1;
        flag = flag + 1;
        flag_counter_neg_1_64 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_1_32 )
        pick_slope(1,slope_index) = -1/32;
        counter_neg_1_32 = counter_neg_1_32 + 1;
        flag = flag + 1;
        flag_counter_neg_1_32 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_1_16 )
        pick_slope(1,slope_index) = -1/16;
        counter_neg_1_16 = counter_neg_1_16 + 1;
        flag = flag + 1;
        flag_counter_neg_1_16 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_1_8 )
        pick_slope(1,slope_index) = -1/8;
        counter_neg_1_8 = counter_neg_1_8 + 1;
        flag = flag + 1;
        flag_counter_neg_1_8 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_one_fourth )
        pick_slope(1,slope_index) = -1/4;
        counter_neg_one_fourth = counter_neg_one_fourth + 1;
        flag = flag + 1;
        flag_counter_neg_1_4 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_one_half )
```

```
        pick_slope(1,slope_index) = -1/2;
        counter_neg_one_half = counter_neg_one_half + 1;
        flag = flag + 1;
        flag_counter_neg_1_2 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_one )
        pick_slope(1,slope_index) = -1;
        counter_neg_1 = counter_neg_1 + 1;
        flag = flag + 1;
        flag_counter_neg_1 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_two )
        pick_slope(1,slope_index) = -2;
        counter_neg_2 = counter_neg_2 + 1;
        flag = flag + 1;
        flag_counter_neg_2 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_4 )
        pick_slope(1,slope_index) = -4;
        counter_neg_4 = counter_neg_4 + 1;
        flag = flag + 1;
        flag_counter_neg_4 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_8 )
        pick_slope(1,slope_index) = -8;
        counter_neg_8 = counter_neg_8 + 1;
        flag = flag + 1;
        flag_counter_neg_8 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_16 )
        pick_slope(1,slope_index) = -16;
        counter_neg_16 = counter_neg_16 + 1;
        flag = flag + 1;
        flag_counter_neg_16 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_32 )
        pick_slope(1,slope_index) = -32;
        counter_neg_32 = counter_neg_32 + 1;
        flag = flag + 1;
        flag_counter_neg_32 = 1;
    end
```

```
    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_64 )
        pick_slope(1,slope_index) = -64;
        counter_neg_64 = counter_neg_64 + 1;
        flag = flag + 1;
        flag_counter_neg_64 = 1;
    end

    if ( overall_segment_edge(overall_segment_edge_index) == seg-
ment_edge_neg_128 )
        pick_slope(1,slope_index) = -128;
        counter_neg_128 = counter_neg_128 + 1;
        flag = flag + 1;
        flag_counter_neg_128 = 1;
    end

    if (flag > 1)
        flag_counter=[ flag_counter_neg_128 flag_counter_neg_64
flag_counter_neg_32 flag_counter_neg_16 flag_counter_neg_8
flag_counter_neg_4 flag_counter_neg_2 flag_counter_neg_1
flag_counter_neg_1_2 flag_counter_neg_1_4 flag_counter_neg_1_8
flag_counter_neg_1_16 flag_counter_neg_1_32 flag_counter_neg_1_64
flag_counter_neg_1_128 flag_counter_1_128 flag_counter_1_64
flag_counter_1_32 flag_counter_1_16 flag_counter_1_8 flag_counter_1_4
flag_counter_1_2 flag_counter_1 flag_counter_2 flag_counter_4
flag_counter_8 flag_counter_16 flag_counter_32 flag_counter_64
flag_counter_128];
        multiple_equal_widths =[segment_count_array; flag_counter];

        while( mult_equal_seg_index <= 30 )

            if(multiple_equal_widths(2,mult_equal_seg_index) == 1)

pick_slope(pick_slope_row,slope_index)=multiple_equal_widths(1,mult_equ
al_seg_index);
               pick_slope_row = pick_slope_row + 1;
            end

           mult_equal_seg_index = mult_equal_seg_index + 1;
        end

       slope_error_segment(slope_error_segment_index) = slope_index;
       slope_error_segment_index = slope_error_segment_index + 1;
       size_slope_error=size(slope_error_segment);
       size_slope_error=size(slope_error_segment,2);

    end
    flag_counter_neg_128 = 0;
    flag_counter_neg_64 = 0;
    flag_counter_neg_32 = 0;
    flag_counter_neg_16 = 0;
    flag_counter_neg_8 = 0;
    flag_counter_neg_4 = 0;
    flag_counter_neg_2 = 0;
    flag_counter_neg_1 = 0;
```

```
flag_counter_neg_1_2 = 0;
flag_counter_neg_1_4 = 0;
flag_counter_neg_1_8 = 0;
flag_counter_neg_1_16 = 0;
flag_counter_neg_1_32 = 0;
flag_counter_neg_1_64 = 0;
flag_counter_neg_1_128 = 0;
flag_counter_1_128 = 0;
flag_counter_1_64 = 0;
flag_counter_1_32 = 0;
flag_counter_1_16 = 0;
flag_counter_1_8 = 0;
flag_counter_1_4 = 0;
flag_counter_1_2 = 0;
flag_counter_1 = 0;
flag_counter_2 = 0;
flag_counter_4 = 0;
flag_counter_8 = 0;
flag_counter_16 = 0;
flag_counter_32 = 0;
flag_counter_64 = 0;
flag_counter_128 = 0;
flag = 0;
total_segment_ambiguity = 0;
pick_slope_row = 1;
mult_equal_seg_index = 1;



right_end = overall_segment_edge(overall_segment_edge_index);

left_end_1_128 = right_end;
right_end_1_128 = left_end_1_128 + 1;

left_end_1_64 = right_end;
right_end_1_64 = left_end_1_64 +1;

left_end_1_32 = right_end;
right_end_1_32 = left_end_1_32 + 1;

left_end_1_16 = right_end;
right_end_1_16 = left_end_1_16 + 1;

left_end_1_8 = right_end;
right_end_1_8 = left_end_1_8 + 1;

left_end_one_fourth = right_end;
right_end_one_fourth = left_end_one_fourth +1;

left_end_one_half = right_end;
right_end_one_half = left_end_one_half +1;

left_end_one = right_end;
right_end_one = left_end_one +1;
```

```
left_end_two = right_end;
right_end_two = left_end_two +1;

left_end_4 = right_end;
right_end_4 = left_end_4 + 1;

left_end_8 = right_end;
right_end_8 = left_end_8 + 1;

left_end_16 = right_end;
right_end_16 = left_end_16 + 1;

left_end_32 = right_end;
right_end_32 = left_end_32 + 1;

left_end_64 = right_end;
right_end_64 = left_end_64 +1;

left_end_128 = right_end;
right_end_128 = left_end_128 + 1;

left_end_neg_1_128 = right_end;
right_end_neg_1_128 = left_end_neg_1_128 + 1;

left_end_neg_1_64 = right_end;
right_end_neg_1_64 = left_end_neg_1_64 +1;

left_end_neg_1_32 = right_end;
right_end_neg_1_32 = left_end_neg_1_32 + 1;

left_end_neg_1_16 = right_end;
right_end_neg_1_16 = left_end_neg_1_16 + 1;

left_end_neg_1_8 = right_end;
right_end_neg_1_8 = left_end_neg_1_8 + 1;

left_end_neg_one_fourth = right_end;
right_end_neg_one_fourth = left_end_neg_one_fourth +1;

left_end_neg_one_half = right_end;
right_end_neg_one_half = left_end_neg_one_half +1;

left_end_neg_one = right_end;
right_end_neg_one = left_end_neg_one +1;

left_end_neg_two = right_end;
right_end_neg_two = left_end_neg_two +1;

left_end_neg_4 = right_end;
right_end_neg_4 = left_end_neg_4 + 1;

left_end_neg_8 = right_end;
```

```
    right_end_neg_8 = left_end_neg_8 + 1;

    left_end_neg_16 = right_end;
    right_end_neg_16 = left_end_neg_16 + 1;

    left_end_neg_32 = right_end;
    right_end_neg_32 = left_end_neg_32 + 1;

    left_end_neg_64 = right_end;
    right_end_neg_64 = left_end_neg_64 +1;

    left_end_neg_128 = right_end;
    right_end_neg_128 = left_end_neg_128 + 1;

    overall_segment_edge_index = overall_segment_edge_index + 1;
    slope_index = slope_index + 1;

end
segment_edge_vs_slope =[overall_segment_edge; pick_slope];

size_of_segment_edge = size(overall_segment_edge);
size_of_segment_edge = size_of_segment_edge(2);
element = 1;
segment_width_index = 1;

while (element < size_of_segment_edge)
    segment_width(segment_width_index) = overall_segment_edge(element +
1) - overall_segment_edge(element);
    element=element + 1;
    segment_width_index = segment_width_index + 1;
end
    segment_width = [overall_segment_edge(1) segment_width];
    segment_width_vs_slope = [segment_width; pick_slope];

%calculate constant
size_overall_segment_edge = size(overall_segment_edge);
size_overall_segment_edge = size_overall_segment_edge(2);
test_index = 1;
index = 1;
constant_segment_index = 1;
constant_index = 1;

constant_segment=curve(1);
while ( index <= size_overall_segment_edge)
    while ( curve(test_index) ~= curve(overall_segment_edge(index)) )
        constant_segment(constant_segment_index) = [curve(test_index)];
        test_index = test_index + 1;
        constant_segment_index = constant_segment_index + 1;

        %curve(overall_segment_edge(index))
        %test_index
    end
    y_constant(constant_index) = min(constant_segment) +
(max(constant_segment) - min(constant_segment))/2;
```

```
        constant_segment;
        constant_segment = 0;
    index = index + 1;
    constant_segment_index = 1;
    constant_index = constant_index + 1;
    index;
end
x_constant_index = 1;
new_segment_edge = [1 overall_segment_edge];
size_segment_edge = size(new_segment_edge);
size_segment_edge = size_segment_edge(2);

while(x_constant_index < size_segment_edge)
    x_constant(x_constant_index) =
x(new_segment_edge(x_constant_index)) +
((x(new_segment_edge(x_constant_index + 1)) - x(new_segment_edge(
x_constant_index )))/2);
    x_constant_index = x_constant_index + 1;
end

constant = y_constant - pick_slope(1,1:size_overall_segment_edge) .*
x_constant;


slope_with_constant=[pick_slope;constant;overall_segment_edge];

new_segment_edge_index = 1;

while(new_segment_edge_index <= size_overall_segment_edge)
    x_approx = lin-
space(x(new_segment_edge(new_segment_edge_index)),x(new_segment_edge(ne
w_segment_edge_index + 1)),new_segment_edge(new_segment_edge_index +
1));
    y = pick_slope(1,new_segment_edge_index).*x_approx + con-
stant(new_segment_edge_index);
        figure(1)
        title('Power-of-2 Slope Approximation')
        plot(x_approx,y)
        hold all
    new_segment_edge_index = new_segment_edge_index + 1;
end

for (check_plot_index = 1:( overall_segment_edge_index - 1))
    check_plot(check_plot_index) =
curve(overall_segment_edge(check_plot_index));
end

    %plot(x,curve)

    figure(2)
    stem(x(overall_segment_edge),check_plot,'g')
    hold all
    plot(x,curve)
    title('Segmentation of curve')
```

```
fprintf('\n slope used vs. how many time used')
fprintf('\n slope -128    = %i',counter_neg_128)
fprintf('\n slope -64     = %i',counter_neg_64)
fprintf('\n slope -32     = %i',counter_neg_32)
fprintf('\n slope -16     = %i',counter_neg_16)
fprintf('\n slope -8      = %i',counter_neg_8)
fprintf('\n slope -4      = %i',counter_neg_4)
fprintf('\n slope -2      = %i',counter_neg_2)
fprintf('\n slope -1      = %i',counter_neg_1)
fprintf('\n slope -1/2    = %i',counter_neg_one_half)
fprintf('\n slope -1/4    = %i',counter_neg_one_fourth)
fprintf('\n slope -1/8    = %i',counter_neg_1_8)
fprintf('\n slope -1/16   = %i',counter_neg_1_16)
fprintf('\n slope -1/32   = %i',counter_neg_1_32)
fprintf('\n slope -1/64   = %i',counter_neg_1_64)
fprintf('\n slope -1/128  = %i',counter_neg_1_128)
fprintf('\n slope 1/128   = %i',counter_1_128)
fprintf('\n slope 1/64    = %i',counter_1_64)
fprintf('\n slope 1/32    = %i',counter_1_32)
fprintf('\n slope 1/16    = %i',counter_1_16)
fprintf('\n slope 1/8     = %i',counter_1_8)
fprintf('\n slope 1/4     = %i',counter_one_fourth)
fprintf('\n slope 1/2     = %i',counter_one_half)
fprintf('\n slope 1       = %i',counter_1)
fprintf('\n slope 2       = %i',counter_2)
fprintf('\n slope 4       = %i',counter_4)
fprintf('\n slope 8       = %i',counter_8)
fprintf('\n slope 16      = %i',counter_16)
fprintf('\n slope 32      = %i',counter_32)
fprintf('\n slope 64      = %i',counter_64)
fprintf('\n slope 128     = %i',counter_128)
fprintf('\n total segments used = %i \n',size_of_segment_edge)
fprintf('\n The total number of segments with segments with multiple
slopes = %i\n',size_slope_error)
fprintf('\n Multiple slopes where located at the following segment:
Segment %i',slope_error_segment)
fprintf('\n')
```

BIBLIOGRAPHY

[1] T. Sasao, J. T. Butler, and M. Riedel, "Application of LUT Cascades to Numerical Function Generators," *Synthesis And System Integration of Mixed Information Technologies 2004*, Kanazawa, Japan, Oct. 18-19, 2004.

[2] B. Parhami, *Computer Arithmetic, Algorithms and Hardware Designs*, pp. 361-367, Oxford University Press, Inc., New York, 2000.

[3] Hao-Yung Lo, Hsiu-Feng Lin and Kuen-Shiuh. Yang, **"**A New Method of Implementation of VLSI CORDIC for Sine and Cosine Computation," *IEEE International Symposium on Circuits and Systems*, Vol. 3, pp. 1984 - 1987, April - May 1995.

[4] Tomas Lang and Elisardo Antelo, **"**CORDIC-Based Computation of ArcCos and ArcSin," *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 132 – 143, July 1997.

[5] Ramesh C. Agarwal, Fred G. Gustavson and Martin S. Schmookler, **"**Series Approximation Methods for Divide and Square Root in the Power3^{TM} Processor," *Proceedings of 14th IEEE Symposium on Computer Arithmetic*, pp. 116 - 123 , April 1999.

[6] D. H. Douglas and T. K. Peucker, "Algorithms for the Reduction of the Number of points Required to Represent a Line or its Caricature," *The Canadian Cartographer*, Vol. 10, No. 2, pp. 112-122, 1975.

[7] http://geometryalgorithms.com/Archive/algorithm_0205, last accessed June 2005.

[8] J. T. Butler, private conversation, Naval Postgraduate School, April 26, 2005.

[9] J. Detrey and F. deDinechin, "Second Order Function Approximation Using a Single multiplication on FPGAs," J. Becker, M. Platzner, and S. Vernalde (eds.), *Proceedings of 14th International Conference on Field Programable Logic and Computer Architecture*, pp. 221-230, Springer-Verlag, Berlin, 2004.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

4. Prof. Jon T. Butler
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

5. Prof. Phillip E. Pace
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA

6. Zaldy M. Valenzuela
   San Diego, CA